# DFspliter: Data Flow Oriented Program Partitioning Against Data Stitching Attacks

Chenyu Zhao[1,2](✉) and Hao Han[1,2](✉)

[1] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China
{cyzhao,hhan}@nuaa.edu.cn
[2] Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing 211106, China

**Abstract.** Sensitive data disclosure caused by attacks is always a serious problem in the field of information security. It has been one of the focuses of researchers' attention in recent years. As defense solutions against control-flow hijacking attacks widely deployed, an attack method targeting non-control data named data stitching attack was developed. It can mount significant damage on applications but difficult to be detected or defended since it does not change control flow of the victimized program. For the purpose of protecting programs from data stitching attack, we propose DFspliter, an automated program partitioning method focusing on data flows. It can protect any C program by dividing the monolithic program into several blocks. Each block runs in a exclusive process. If attacks cause data leakage in a block, only the data in this block might be stolen, while the data in other blocks will not be affected. We implement our method in the form of an LLVM plug-in, so that software developers can automatically complete the process with only a few compilation commands. Finally, an example modeled after the real-world vulnerability is used to show the effectiveness of our method.

**Keywords:** Program partitioning · Data stitching attack · Complier plug-in · Data flow

## 1 Introduction

Today, it is not uncommon for applications to be attacked. The flaws in the program itself and the bugs accidentally left will become weaknesses and bring opportunities for attackers. In order to solve this problem, researchers have proposed different solutions. Program partition is one of them. One of the examples is that modern applications such as Google Chrome follows the multi-process model and divide the program into different processes. So errors in one process will not directly affect other processes. Thanks to this, more than 600 security vulnerabilities was detected in the code of Chrome in 2014, but the damage caused by these vulnerabilities is very limited [11]. However, manual program partition will increase the difficulty of development and maintenance, and thus increase the cost. This is not affordable for some development teams.

Among the current attack methods on applications, there is a method called data stitching attack [4]. It is proposed under the background that it is more and more difficult to implement an attack on the control flow. Data stitching attack can stitch together different data flows which are not related in the original program. After making these data flows relevant, the attacker can use the original output in the program to obtain some sensitive data that should not be output without modifying the control flow. Current data protection methods are mostly focusing on control flow, so the defense effect against such attacks is not satisfactory.

With the goal of defending against data splicing attacks, we propose an automated program partitioning method called DFspliter. Developers do not need to consider security when writing code. They only need to do is adding a few commands when compiling. Our method will divide the program into several program blocks, and different program blocks run in different process spaces. Our partitioning method will prioritize safety, and followed by efficiency. Once a program block is attacked, only the data in the attacked block may be tampered with or stolen. The data in other blocks will not be affected. Because the low data coupling between different program blocks, it will be difficult for the attacker to calculate data that has not been leaked based on part of the leaked data.

Main contributions of this paper are as follows:

– DFspliter will focus on the transfer of data between instructions on the basis of the program dependency graph, and analyze this data flow. We divide the data flows with strong correlation into the same block as much as possible, and divide the data streams with weak correlation into different blocks as much as possible.
– We partition the program on the function level. Considering the additional communication overhead caused by passing parameters and other reasons after partitioning, our method will try to reduce the communication overhead by merging some program blocks while having no significant negative impact on security.
– We plan to implement the proposed method in the form of LLVM Pass. Processes communicate through pipes with well-defined interfaces, and the original semantics of the program does not change before and after segmentation.
– Our method will minimize the artificial component, users do not need to rewrite a lot of code, only need to use LLVM to load the required Pass to automatically complete the segmentation. However, considering the complexity of pointers and environmental variables, some manual work may be required.

The rest of this paper is organized as follows. Section 2 is related works. Section 3 is the design of our method, which introduces the program partitioning method proposed in this paper in two parts. Section 4 is the implementation, which introduces the preliminary implementation of our method based on LLVM. Section 5 is the system verification, which shows the effect of applying a prototype of our method to a sample program. Section 6 is conclusion.

## 2 Related Work

Recently, researches on program segmentation can be divided into two directions. One direction is privilege splitting. Privileges are generally expressed as system calls used by the program. These methods restrict the system calls that can be used by each program block after segmentation, reducing the privilege obtained by the attacker after a successful attack. The other direction is protecting sensitive data, such as user passwords, user privacy, private keys, etc. The common method of protecting sensitive data is separating the part that containing sensitive data from the original program, with targeted protection through various methods such as encryption and verification.

Some methods require a lot of manual operation. These tools provide runtime analysis results of the program and interfaces required for manual segmentation. The developer understands the privileges required by each part of the source code according to the runtime analysis results of the program. Then use specifically implemented interface in operating systems to achieve segmentation, such as Wedge [1]. These methods save the workload of analysis for the developer, but the remaining workload is still considerable. In addition, Wedge also needs to add special primitives in the operating system to support program partitioning. Although it is relatively easy to implement in the open source operating systems such as Linux, there will be difficulties in implementation in systems such as Windows. Modifying the program by developer will also request the ability of programming, which may affect the efficiency of the modified program.

When automation level increased, program partitioning tools require developers to use comments to mark the required privileges in the source code, or manually distinguish the privileges obtained from automatic analysis. The main purpose of Trellis [10] is to provide different privilege for different users. Privtrans [2] and ProgramCutter [14] are intended to limit the set of system calls that the program can use to avoid giving the attacker excessive privileges when the program is attacked. All these three tools mentioned above can automatically work based on the specific comments or the manual classification of privileges written by the developer. These tools also include process isolation or monitoring. Among them, Trellis needs to add special system calls to the operating system, and perform operations such as lifting privilege through the system call, and monitoring program and its system call. Privtrans and ProgramCutter need to modify a series of function calls that cross boundaries, replacing each ordinary function call with IPC (Inter Process Communication) or RPC (Remote Procedure Call). But this process cannot be completed automatically, especially for those complex data structures defined by developers. They still need to manually write interfaces, and tools can simplify this part of the work.

Some tools are almost completely automatic. These tools, such as Trapp [12, 13], can automatically identify different privileges according to the calling situation of library functions, and perform the partitioning highly automatically based on a small number of preset parameters. The effect of automatic partitioning may not be as good as those with manual intervention. Trapp uses IPC to communicate, and its performance overhead is generally not obvious in the test program. But if the split location is just in a high-frequency function call, it may bring significant performance overhead. The main problem of automatic privilege splitting is the difficulty to take into account the implied privilege requirements in operating environments and configuration files, which may cause errors in the modified program. To solve this problem, it is inevitable to add manual interference.

Program partitioning technology in the direction of protecting sensitive data generally relies on developers' manual work. These methods generally needs programmers to marking the location where sensitive data is generated or where sensitive data has died. The tool can analyze and rewrite the program according to the annotations in the source code and the program dependency graph. Then protect the designated sensitive data in a targeted manner. In addition, some methods such as MPI [9] require developers to explain the main data structure. Then the program is divided accordingly to fit the high-level design of the program, and separate different data more naturally. Some solutions, such as Glamdring [5] and SeCage [8], use hardware-provided features such as Intel SGX to rewrite programs. They have excellent defense capabilities against attacks, but these methods are obviously hardware-related, so it may be difficult to migrate to different hardware architectures.

Solutions that only rewrite the program itself, such as Program-mandering [7] and PtrSplit [6], use complex methods to deal with complex parameters, especially the structure of C/C++ that contains pointers and allocates memory through `malloc`. Improper handling may cause overly complex IPC or RPC and affect the performance of the partitioned program. But the performance problem is expected to be solved through more diversified indicators. And the effectiveness of partitioning can be improved at the same time. Some methods such as PtrSplit and Program-mandering only divide the program into two program blocks and protect one of them. This may cause the protected part to be too large in some cases, which may result in a decrease in security or operating efficiency. However, these methods are expected to be extended to multiple partitions to make up for this defect.

## 3 System Design

Our method splits applications into multiple blocks at function level by inferring weakly correlated data flows represented by sequences of instructions that operate on the same data in Program Dependence Graph (PDG) [3]. Each block will run in a separate process and function calls between blocks are achieved by well-defined interfaces. Through separation, the compromise of one block does not directly lead to the compromise of other blocks. Many modern applications such as Chrome browser are designed in this distributed style to increase security, but there is no automated methods to partition legacy programs based on the separate set of data flows inferred from those programs.

The outline of our system is shown in Fig. 1. It takes the source code of a monolithic C program as input. Firstly, it establishes PDG based on the source program. Then it detects data flows based on PDG and calculates the partitioning scheme. Finally, it modifies the input code to implement segmentation. It is difficult to conduct a pure automated system including code analysis and code modification, we just implemented a prototype to establishing PDG and calculating partition scheme.

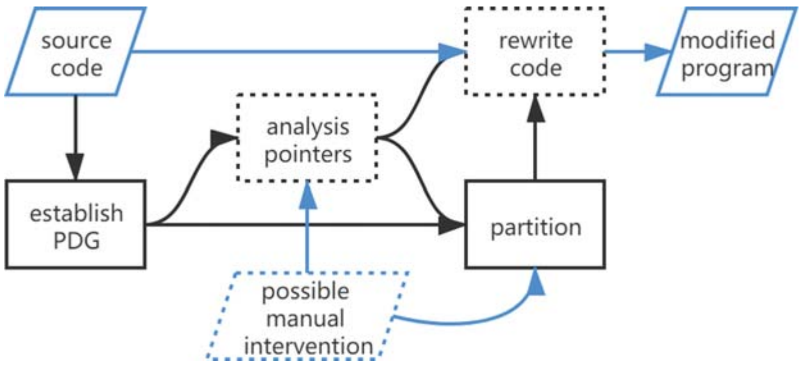

**Fig. 1.** System outline.

## 3.1 Motivation Example

First, we use a running example as follows as an example to explain data stitching attack. It is modeled after a web server. Firstly, it loads a private key from a file to establish HTTP connection. After receiving a connection from client, it reads the message received and sanitizes it by calling `checkInput`. Then, the program calls `getFile` to read the content of the file and send it to the client.

Before loading the content of the file, the program uses `strcat` to get the full path. If the string `reqFile` has enough length to cause overflow at `strcat`, the second pointer `reqFile` will be changed. As long as the attacker chooses a message with appropriate length, the second pointer can point to the address where private key `privKey` stored. Then the private key will be sent to the client as a part of output. The data flow of the private key and the data flow of the input file name has no intersection. They have no dependence on each other and have no shared memory. But the attacker can force the `reqFile` point to the private key by causing overflow with out changing the control flow.

```
void getFile(char *reqFile, char *output){
    char fullPath[BUFSIZE] = "/path/to/root/", output[
        BUFSIZE];
    strcat(fullPath, reqFile);//stack buffer overflow
    result = retrieve(fullPath);
    sprintf(output,"%s:%s",reqFile,result);
}
int server(){
    char *userInput, *privKey, *result, output[BUFSIZE];
    privKey = loadPrivKey("/path/to/privKey");
    GetConnection(privKey, ...);//HTTPS connection using
        privKey
    userInput = read_socket();
    if (checkInput(userInput)) {//user input OK, parse
        request
        getFile(getFileName(userInput),output);
        sendOut(output);
    }
}
```

To defend against this attack, a feasible method divides the program into two parts running in different processes. The function `server` is executed in one part and the function `getFile` is executed in the other. Only the process that runs `server` has the private key stored in it memory, so the attacker can not get it in the other process and the security is enhanced.

However, the source code of real-world programs usually not available for attackers. So what will be attacked are binary programs. And the attack is hard to predict from source code alone. In this situation, we aim to provide protection for all data flows rather than one or two of them. We analyze the program to get sequences of instructions that operate on the same data in succession with limited modification times. Then we partition the source program into several blocks running in different processes and minimize the sequences of instructions between blocks. After that, if one of those blocks is attacked, only the data used in the victimized block may be stolen and data in other blocks which is the majority is not affected.

## 3.2 Design of PDG

In the method proposed in this paper, most of the analysis and processing steps need to be based on the PDG. Therefore, establishing the program dependency graph according to the input code is the first step of the entire processing flow.

The program dependency graph was first proposed in 1984. It is a directed graph that intuitively describes the data dependency and control dependency in the program. It is the basis of many program analysis methods and code optimization methods. Each point of PDG corresponds to a statement of high-level language or an instruction in assembly language. The edge of PDG represents the dependency between instructions or statements, which can be divided into control dependency and data dependency. Among them, data dependency can be divided into two cases: def-use dependency and read after write dependency.

If the execution of instruction I1 is determined by another instruction I2, I1 control depends on I2. If the first executed instruction I1 and the later executed instruction I2 exchange execution sequence, the result may change, then I2 data depends on I1.

According to the basic definition of PDG, we build nodes corresponding to all instructions, global variables, and parameters of functions and calls in the program.

Control dependent edges and data dependent edges are built within the function. The control dependent edge is from the jump instruction node to each instruction node in all the basic blocks controlled by it to decide whether to execute. For the def-use dependency, if an operand of one instruction is the result of another instruction, an edge will be built from the instruction node that provides the result to the instruction node that uses the result. For the read-after-write dependency, if two instructions operate on the same part of memory, an edge will be built from the instruction node that writes the memory to the instruction node that reads the memory.

On the basis of the definition of PDG, in order to pertinently handle function calls, two nodes in different but related functions are divided into the following 4 cases. These dependent edges need to be built separately.

– If a function is called, build an edge from the actual parameter node to the formal parameter node.
– If a function has a return value, build an edge from the return value in the called function to the return value in the caller function.
– If an instruction reads a global variable, build an edge from the global variable to the instruction.
– If an instruction writes a global variable, build an edge from the instruction to the global variable.

After the dependency of call and return is processed as above, the return value in the caller function will not be directly connected to the actual parameters.

## 3.3 Partitioning Method

The data that can be obtained by data stitching attack must be in the memory when the program runs to the location of the vulnerability. Therefore, we divide the program into multiple blocks. Different blocks contain different sets of functions and runs in different processes. This can reduce the data that may be leaked when under attack, thereby improves the security of the program. In order to defend data splicing attacks better, we should make each piece of data to appear in as few program blocks as possible after partitioning. In addition, the data that has been modified multiple times is generally weakly associated with the original data, and they can be regarded as different data. We define instruction flow as follows to represent data flow with limited modifications:

**Definition 1.** *A set of a chosen instruction and instructions that may operate on the same data before or after the execution of the chosen instruction with modification times on the path less or equal than $K$.*

For convenience, we call an instruction flow by the name of the chosen instruction. In the definition, $K$ is an adjustable parameter, which is used to indicate how many times a piece of data has been modified can be regarded as completely different data. The smaller the K, the stricter the requirement. The larger the $K$, the more ambiguous the requirement. In the current prototype system, we set $K$ to 5.

In programs, searching for the instruction flow for each instruction as the chosen instruction may cause greater time penalty and lots of repeated calculations. Considering that the granularity of our partitioning is at the function level, the partitioning process mainly focuses on the instruction flow between each pair of functions, rather than processing every instruction. Accordingly, the instruction flow between functions can be defined as follows:

**Definition 2.** *The instruction flow set $ds(f_A, f_B)$ from function $f_A$ to function $f_B$ is a set of instruction flows that take one of the instructions of $f_A$ that store parameters, set arguments of call instructions, and set return values as the chosen instruction and contain at least one instruction in $f_B$.*

For example, in the program shown in Sect. 3.1, $ds(\texttt{server}, \texttt{getFile})$ has the instruction flows starting from `userInput` and `output` in `server`.

We then calculate the optimal partitioning based on the relevance between blocks, the complexity of each block, and the time penalty.

**Relevance.** The instruction flow in the program generally has different degrees of importance. Among them, the widely used but basically unchanged data may be more important because it may play a controlling role or be a parameter of multiple calculation processes. For each instruction flow $x$, the number of instructions in it is defined as its influence range $ir(x)$, which is used to describe the importance of the instruction flow. Then the relevance $R(f_A, f_B)$ of function $f_A$ and $f_B$ can be defined as follows:

$$R(f_A, f_B) = \max_{i \in ds(f_A, f_B)} ir(i) + \max_{j \in ds(f_B, f_A)} ir(j) \quad (1)$$

In our example shown in Sect. 3.1, $R(\texttt{server}, \texttt{getFile})$ is eaual to $ir(\texttt{userInput})$ because $ir(\texttt{userInput})$ is larger than $ir(\texttt{output})$ and `getFile` has no return value.

**Complexity.** Even if the program is divided into multiple processes for execution, the program blocks that are directly attacked may still leak the data it contains. Therefore, it is necessary to control the size of each block to prevent it from leaking too much data or contains too many weaknesses when it is attacked. We define the complexity of a program block $b$ as $C(b)$. To achieve a higher automation level, we do not request programmer to provide annotation information. That means we do not have clearly indicating where the program has a bug or where data leakage may occur. So complexity may be the total number of lines in source code, the number of instructions and other indicators that make sense.

**Time Penalty.** After the source program rewritten into a multi-process program, some function calls will be rewritten as inter-process communication (IPC), and data required for program execution such as parameters and return values should be passed between processes, which will cause additional time penalty. For any function $f_A$, define the time penalty $T(f_A)$ as the time required to call the function through IPC. Then define the set of functions called through IPC as $G$. In our method, an IPC contributes 100 times time penalty than a byte transmitted between processes. These values are based on our test of IPC performance under the Linux system.

Priority should be given to security during partitioning, so relevance is the most important indicator. Complexity and time penalty are secondary indicators. Define $B(f_A)$ as the block where function $f_A$ is located. Then during the partitioning, the following expression needs to be minimized:

$$k_1 * \sum_{B(i)!=B(j)} R(i,j) + k_2 * \max_k C(k) + k_3 * \sum_{l \in G} T(l) \tag{2}$$

Among them, $k_1$, $k_2$, and $k_3$ are adjustable parameters, which we will determine in future work, or leave them as user-set parameters. In our example, there is only two functions, so $R(\texttt{server}, \texttt{getFile})$ is the largest. Then, the two functions will be divided into two blocks. And the sensitive data in `server` will be separated from the vulnerable `getFile`.

## 4 Implementation

We implement the major part of our method in the form of LLVM pass under the framework of the open source project LLVM. This section will introduce the realization of our method described in Sect. 3 in two parts.

LLVM is a collection of modular, reusable compiler and tool chain technology. It has a special intermediate language called LLVM IR. The front end of LLVM is responsible for translating high-level languages into LLVM IR. The most common front end is the compiler Clang. There are various passes in the middle, and each pass is a specific function that can be applied to LLVM IR. The back end is responsible for assembling LLVM IR into machine code that can run on target machine. We implement our method as a series of passes with a total of 2578 lines of code.

### 4.1 Implementation of PDG

We establish PDG based on LLVM IR and implements it in the form of an LLVM pass. There is no strict sequence requirement for the establishment of PDG, which basically conforms to the description in Sect. 3.2.

In the process of creating nodes, we build nodes for instructions and global variables respectively. For call instructions in LLVM IR, We build special call nodes and connect them with corresponding parameter trees.

When analyzing control dependence, we use the post dominator tree built in LLVM. When analyzing read-after-write dependence, we use alias analysis built in LLVM to improve accuracy. When analyzing the def-use dependence, thanks to the static single assignment (SSA) feature of LLVM that each variable assigned as the result of an instruction can be assigned only once, where does an operand comes from can be determined easily.

To handle inter-procedural dependence, we build parameter trees [6] instead of a single node for every formal parameters, actual parameters, and return values. The parameter trees include detailed information about structures, arrays and pointers based on the type of parameters. Edges are built with nodes on parameter trees to represent inter-procedural dependence and read-after-write dependence.

Some call instructions in the program may call functions through function pointers. In this situation, LLVM IR cannot determine the called function. The analysis of function pointers is a complicated task, and we are not focusing on it now. If this happens, our method will try to match the possible called functions by type, and build a corresponding dependent edge for each function that meets the conditions.

## 4.2 Implementation of Partitioning

In our implementation, the function nodes in PDG are colored to represent the partitioning scheme. Each color represents a block of the modified program. We implemented this in the form of an LLVM Pass.

First, we process the corresponding points in the PDG of the formal parameters, actual parameters, and return values in the function. According to the definition of the instruction flow, the forward tracking or reverse slicing is performed with these points as the chosen point to obtain the functions that instruction flows can reach and instruction flows' influence range. It should be noted that forward tracking and reverse slicing are only performed along the data dependent edges, and do not involve control dependent edges. These are also limited by the number of data modification $K$. After that, each pair of functions is processed and the relevance between them is calculated.

What we have implemented is a simple prototype system, without considering the complexity of program blocks and time penalty. So only security is considered when coloring, that is, the relevance between functions.

At the beginning of the partitioning process, all functions are located in separate program blocks. For a program with $n$ functions, the algorithm continues to merge the program blocks according to the relevance between the functions until the number of program blocks does not exceed $\log_2 n + 1$. For each step of merging, each pair of program blocks is scanned, and the sum of the relevance between the functions located in it is calculated. Then in each step of merge, the algorithm selects a pair of program blocks with the largest sum of relevance and merge them into one block.

## 5 Evaluation

### 5.1 Validation of DFspliter

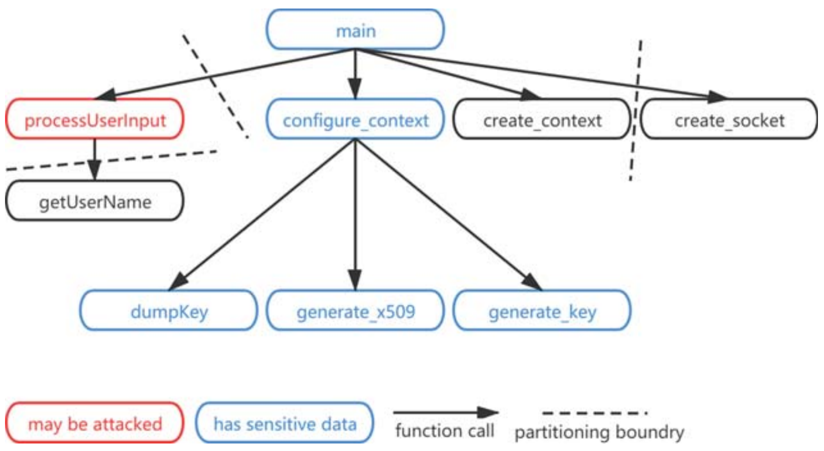

**Fig. 2.** Partitioning result of ssl_server.

We use a C program named `ssl_server` as a test program (Fig. 2). Part of the program is shown as follows. It is a vulnerable server program based on open SSL and has a corresponding attack program `ssl_client` with it. `ssl_server` generates a pointer `ctx` in the `main` function by calling the `create_context` function, which has a wide range of influence. After that, a private key `pkey` for encrypted communication is generated by calling `configure_context` in the function `main` and stored in the memory pointed to by `ctx`. After everything is ready, the main function cyclically receive messages from the client, and processes and responds through the `processUserInput` function. There are vulnerabilities in the `processUserInput` function that can be exploited by data stitching attack. When the message length is appropriate, the private key may be sent back to the client as a message.

```
void configure_context(SSL_CTX *ctx){
    /* Generate private key pkey */
    SSL_CTX_use_PrivateKey(ctx, pkey);
}
char buf[1024];
char* processUserInput(char *input, int size){
    char *yourName, greeting[16] = "Welcome ";
    yourName = getUserName(input);
    strcat(greeting, yourName);
    sprintf(buf, "%s: %s", yourName, greeting);
    return buf;
}
int main(int argc, char **argv){
    /* Prepare for connection */
    ctx = create_context();
    configure_context(ctx);
    sock = create_socket(4433);
    while(1) {
        /* Accept a request and start connection */
        bytes = SSL_read(ssl, buf, sizeof(buf));
        char *output = processUserInput(buf, bytes);
        SSL_write(ssl, output, strlen(output));
        /* End connection */
    }
    /* Clean up */
}
```

We use the prototype described in Sect. 4 to strengthen `ssl_server`, and rewrite the source program into a multi-process program according to the partitioning result. The program is partitioned into 4 blocks that run in different processes and communicate with each other through named pipes. First 3 blocks contains `create_context`, `processUserInput` and `getUserName` respectively. The function containing the sensitive data `pkey`, including the main function, is located in the fourth block. Since there is no `pkey` stored in the process where the vulnerable `processUserInput` is located, the private key can no longer be obtained through data stitching attacks, and the security of the program is improved.

## 5.2 Performance Overhead

We use another C program to test the communication overhead after partitioning. Part of this program is shown as follows. This program has a function named `refuse`, which is called by `main` when the input is illegal. What the function `refuse` does is opening a file, writing a constant string into the file and closing the file. To test the overhead, we change the type and the number of parameters of `refuse` and rewrite the program into a 2-process program which divides the two functions into different processes. We always provide an illegal input and let `main` calls `refuse` 1000 times. The results are shown in Table 1.

```
void refuse(int y){//Parameter may be changed
    f=fopen("/* File name */","w");
    fprintf(f,"do not support argument y=%d !\n",y);
    fclose(f);
    return;
}
int main(int argc,char** argv){
    int x,y,z;
    scanf("%d%d",&x,&y);
    for(int i=0;i<1000;++i)
        if(y>=0){/* Do something */}
        else refuse(y);
    return 0;
}
```

**Table 1.** Communication Overhead of different parameters

| Parameters type in C | Number of parameters | Time spent before partitioning (ms) | Time spent after partitioning (ms) | Communication Overhead (%) |
|---|---|---|---|---|
| int | 1 | 60.95 | 68.75 | 12.79 |
| int* | 1 | 58.31 | 69.91 | 19.89 |
| int[1000] | 1 | 58.45 | 87.80 | 50.21 |
| int | 10 | 56.90 | 80.10 | 40.77 |

Communication overhead is slightly lower than that of reading and writing files. Inter-process calls have a basic time consumption, so are parameter transmissions. The increase in the number of bytes transferred will also bring additional time cost, but not as much as the number of parameters. Generally, most of the running time is spent on logic of programs rather than calling a function 1000 times, so the communication overhead is acceptable. Our approach also tries to avoid transmissions of parameters with large number of bytes.

## 6 Conclusion

We designed and partially implemented a program partitioning method called DFspliter based on LLVM. Any C language program with single block can be automatically divided into a program consist with multiple program blocks during the compilation process. Each block will execute in a different process and communicates with others through well-defined interfaces. Because different processes have different memory spaces and different data stored in them, potentially sensitive data can be protected. When one block is attacked, it will not affect the security of data in other blocks. Aiming at the characteristics of data stitching attacks, our method uses data flow as the main basis for partitioning procedures, and improves the pertinence of segmentation by splitting different data flow with lower correlation into different program blocks. In order to get better partitioning result, we used a special algorithm to perform coloring operations based on the analysis of data flow. Finally, the input program can be changed into a multi-process program and the security is increased.

**Acknowledgements.** We sincerely thank reviewers for their insightful feedback. This work was supported in part by NSFC Award #61972200.

## References

1. Bittau, A., Marchenko, P., Handley, M., Karp, B.: Wedge: splitting applications into reduced-privilege compartments. In: 5th USENIX Symposium on Networked Systems Design and Implementation, pp. 309–322 (2008)
2. Brumley, D., Song, D.: Privtrans: automatically partitioning programs for privilege separation. In: 13th USENIX Security Symposium, pp. 57–72 (2004)
3. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. (TOPLAS) **9**(3), 319–349 (1987)
4. Hu, H., Chua, Z.L., Adrian, S., Saxena, P., Liang, Z.: Automatic generation of data-oriented exploits. In: 24th USENIX Security Symposium, pp. 177–192 (2015)
5. Lind, J., et al.: Glamdring: automatic application partitioning for Intel SGX. In: 2017 USENIX Annual Technical Conference, pp. 285–298 (2017)
6. Liu, S., Tan, G., Jaeger, T.: PtrSplit: supporting general pointers in automatic program partitioning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2359–2371 (2017)
7. Liu, S., et al.: Program-mandering: quantitative privilege separation. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 1023–1040 (2019)
8. Liu, Y., Zhou, T., Chen, K., Chen, H., Xia, Y.: Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 1607–1619 (2015)
9. Ma, S., Zhai, J., Wang, F., Lee, K.H., Zhang, X., Xu, D.: MPI: multiple perspective attack investigation with semantic aware execution partitioning. In: 26th USENIX Security Symposium, pp. 1111–1128 (2017)
10. Mambretti, A., et al.: Trellis: Privilege separation for multi-user applications made easy. In: International Symposium on Research in Attacks, Intrusions, and Defenses, pp. 437–456 (2016)
11. Rogowski, R., Morton, M., Li, F., Monrose, F., Snow, K.Z., Polychronakis, M.: Revisiting browser security in the modern era: New data-only attacks and defenses. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 366–381 (2017)
12. Trapp, M., Rossberg, M., Schaefer, G.: Program partitioning based on static call graph analysis for privilege separation. In: 2015 IEEE Symposium on Computers and Communication (ISCC), pp. 613–618 (2015)
13. Trapp, M., Rossberg, M., Schaefer, G.: Automatic source code decomposition for privilege separation. In: 2016 24th International Conference on Software, Telecommunications and Computer Networks (SoftCOM), pp. 1–6 (2016)
14. Wu, Y., Sun, J., Liu, Y., Dong, J.S.: Automatically partition software into least privilege components using dynamic data dependency analysis. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 323–333 (2013)