CFHider: Control Flow Obfuscation with Intel SGX

Yongzhi Wang*†, Yulong Shen §*, Cuicui Su §*, Ke Cheng §*, Yibo Yang §*, ANter Faree §*, Yao Liu ‡
* *School of Computer Science and Technology*, *Xidian University*, Xi'an, China
†*Department of Computer Science and Information Systems*, *Park University*, Parkville, MO, USA
‡*Department of Computer Science and Engineering*, *University of South Florida*, Tampa, FL, USA
§*Shaanxi Key Laboratory of Network and System Security*, *Xidian University*, Xi'an, China
ywang@park.edu, ylshen@mail.xidian.edu.cn, ccsu.hannah@gmail.com,
kecheng@stu.xidian.edu.cn, bobyangpopo@gmail.com, anterfaree@stu.xidian.edu.cn, yliu@cse.usf.edu

***Abstract*—When a program is executed on an untrusted cloud, the confidentiality of the program's logics needs to be protected. Control flow obfuscation is a direct approach to obtain this goal. However, existing methods in this direction cannot achieve both high confidentiality and low overhead. In this paper, we propose CFHider, a hardware-assisted method to protect the control flow confidentiality. By combining program transformation and Intel Software Guard Extension (SGX) technology, CFHider moves branch statement conditions to an opaque and trusted memory space, i.e., the enclave, thereby offering a guaranteed control flow confidentiality. Based on the design of CFHider, we developed a prototype system targeting on Java applications. Our analysis and experimental results indicate that CFHider is effective in protecting the control flow confidentiality and incurs a much reduced performance overhead than existing software-based solutions (by a factor of 8.8).**

I. Introduction

Cloud computing enables users renting computing resources on demand and executing their programs remotely. However, when the remote environment is untrusted, protecting the confidentiality of the program logic becomes an important security requirement, especially when the algorithm of the program is an intellectual property. Control flow obfuscation, a method transforming program into an unintelligible format while preserving the functionality, is a straightforward method in protecting the confidentiality of program logics. Existing works primarily focused on software-based transformation. However, those works suffer from low security, low expressiveness, or high overhead. For example, Wang et al. [1] proposed to combine control flow flattening and pointer alias construction techniques. They proved that statically analyzing the proposed obfuscated control flow is an NP-complete problem. However, such a method cannot defeat dynamic analysis [2]. Sharif et al. [3] transformed the branch statement condition into an encrypted format, which can only protect the confidentiality of the "equal" and "not equal" predicates, thus losing generality. Lan et al. [4] proposed to transform the conditional instructions with lambda calculus simulations, which can defeat symbolic execution-based reverse-engineering attacks. However, its experimental results showed a high overhead: when protecting only 30% of conditions, its execution time is 68.5 to 248.1 times higher than that of the original program.

To address the above limitations, in this paper, we propose *CFHider*, a general control flow obfuscation solution that introduces a much reduced performance overhead. CFHider combines program transformation with Intel *Software Guard Extension* (*SGX*). It transforms the condition of each branch statement into a *CFQ* function call and moves its execution into the *enclave* that is considered as an opaque and trusted execution environment. To further obfuscate the control flow, CFHider inserts indistinguishable *fake branch statements* into the program. To defeat dynamic analysis-based attacks, CFHider identifies and introduces *obfuscation invariants* for the branch statement to defeat passive attacks (see Section IV-B3), and introduces *data flow check* in the CF Enclave to defeat active attacks (see Section IV-C3). Our analysis showed that CFHider can defeat static and dynamic, passive and active attacks.

To measure the applicability and performance, we developed a prototype system targeting on Java applications. Our experimental results showed that CFHider can protect all Java (Hadoop) applications that we tested and incurs a much reduced performance overhead than previous software-based methods. For instance, when protecting 100% of branch statement conditions, CFHider incurred an average performance overhead of 20.75% when executing Hadoop applications, and introduced 2.9 to 3.7 times of performance overhead when executing CPU intensive applications. Our analysis showed that, in CFHider, executing one protected branch condition costs $4.31\mu s$. Compared to [4], where the execution time of one protected branch is $38.19\mu s$, our solution has reduced the performance overhead by a factor of 8.8. The reduced overhead is majorly attributed to the fact that SGX `ecalls` combined with our LRU cache (in Section IV-C2) introduced a smaller overhead than software-based simulations [4] [5].

By concentrating on the control flow, CFHider achieves a higher security guarantee and a wider generality compared to other endeavors that moving the entire or part of the binary into SGX enclave, such as Haven [6], Graphene-SGX [7], SCONE [8], and Glamdring [9]. Directly moving the binaries into enclave introduces a large Trusted Computing Base (TCB). While lacking a formal analysis, the vulnerabilities in any of the binaries can breach the defense. On the other hand, solutions such as VC3 [10] targets on a specific system

(e.g., C++ version of MapReduce), thus lacking generality. In contrast, CFHider only moves the control flow to the enclave in a security-proven format (see Section V), thus can achieve a high confidentiality guarantee. Although the prototype system is tested on Java applications, the idea of CFHider can be directly applied to applications written in most programming languages. Our contributions can be summarized as follows.

- We propose a hardware-assisted solution, called CFHider, to protect the control flow confidentiality of the programs that are executed remotely. We proved that CFHider can effectively defeat static and dynamic analysis targeting on restoring the original control flow.
- We developed a prototype system that targets on Java programs. Experimental results show that CFHider is applicable to all Java (Hadoop) programs that we tested and introduces a much reduced performance overhead than previous software-based methods.

II. Preliminaries: Intel SGX

SGX [11] [12] is a set of x86-64 ISA extensions that enables a protected execution environments (called enclaves) without requiring trust in anything but the processor and the code placed inside enclaves. Enclaves are protected by the processor: the processor controls access to enclave memory. Instructions that attempt to read or write the memory of a running enclave from the outside of the enclave will fail. Enclave code can be called from untrusted code by means of a callgate-like mechanism that transfers control to a user-defined entry point, namely *ecall*, inside the enclave. SGX supports *remote attestation*, which enables a remote system to verify cryptographically that specific software has been loaded within an enclave, and establishes shared secrets allowing it to bootstrap an end-to-end encrypted channel with the enclave.

III. The Security Model

We assume the attackers are interested in obtaining the control flow of the program uploaded by the user, i.e., compromising the control flow confidentiality. However, the attackers are not interested in compromising the computation integrity, such as tampering the computation results.

For the environment setting, we assume that the user's local environment is free of attacks. However, the public cloud is untrusted. On the public cloud, we assume the processors support SGX. Yet the software stacks on the public cloud host, such as the hypervisor and the OS, are untrusted. To facilitate our description, we call the enclaves as the *trusted area* and call the software stacks on the public cloud as the *untrusted area*. The attackers can be outside attackers, malicious cloud vendor employees, or malicious users who are co-hosted with benign cloud users. We assume that the attacker can perform both static analysis and dynamic tracing in the untrusted area.

We do not have special restrictions on the programs to be protected. As long as the program itself does not reveal its control flow intentionally (e.g., explicitly printing out the control flow), CFHider will work well.

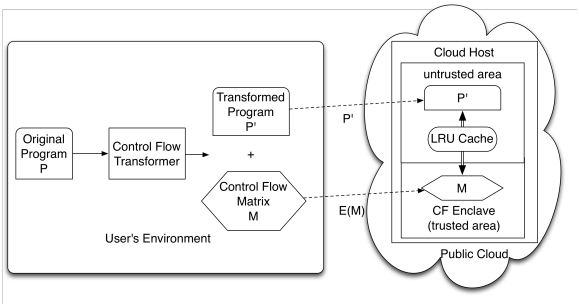


Fig. 1: The architecture of CFHider

Side-channel attack is out of the scope for this paper. Although recent side-channel attacks such as Foreshadow [13], are capable of extracting encryption key of current SGX implementation and could eventually obtain the decryption key of the control flow matrix, as discussed in [13], *"Foreshadow exploits a micro architectural implementation bug, and does not in any way undermine the architectural design of Intel SGX and TEEs in general."*

IV. System Design

A. Architecture

The architecture of CFHider is shown in Fig. 1. It consists of two parts: the trusted user's environment and the untrusted public cloud. For an *original program* P that needs to be executed on the public cloud, CFHider first transforms it into a *transformed program* P' and a *control flow matrix* M on the user's local environment. P' differs from P in that more branch statements are inserted to obfuscate the control flow and the logic of each condition is moved to M. After the transformation, P' and the ciphertext of M, marked as $E(M)$, will be sent to the public cloud: P' will be sent to the cloud host and executed in the untrusted area; $E(M)$ will be sent to the *CF Enclave*, a SGX enclave. The CF Enclave is firstly verified through SGX remote attestation procedure. When passed, the CF Enclave is believed to be trusted. It thus will obtain the decryption key of $E(M)$ to restore M. During the execution of P', the conditions in each branch statement will be evaluated in the CF Enclave based on the content of M. Frequently inquiring CF Enclave will incur a significant performance overhead. To reduce that, the CF Enclave introduces the *LRU Cache*. Each inquiry will first go through the LRU Cache. Only when a cache miss happens, the actual CF Enclave inquiry will be performed.

In the entire execution procedure, M only resides in the user's environment and the attested CF Enclave. Therefore, it does not directly leak condition information to the attacker. With the global picture in mind, we will introduce the program transformation details, followed by the CF Enclave design.

B. Program Transformation

We first describe the basic scheme that transforms each condition in the original program. Based on this, we describe *fake branch insertion*, which further obfuscates the control flow.

Finally, we introduce *obfuscation invariants*, which defeats dynamic passive attacks.

1) Basic Scheme: The control flow transformation is performed on the *3-address code* [14], an intermediate representation of the program. Each 3-address code instruction has at most three operands and is typically a combination of an assignment and a binary operator. Without losing generality, the 3-address code of a branch statement is in the following format:

$$\text{if } (x \ op \ y) \text{ then } \{\text{goto L}\} \tag{1}$$

, where x and y are two variables in the program. op is one of the six possible operators, defined as:

$$op ::= > \ | \ < \ | \ == \ | \ != \ | \ >= \ | \ <= \tag{2}$$

$x \ op \ y$ is the *condition* of the branch statement. L is the address of the *target statement* that the control flow will jump to if $x \ op \ y$ is evaluated to `true`.

The basic scheme of the transformation is performed as follows. For each branch statement s defined as (1), we define the set $U(s) = \{x, y\}$, which contains variables appear in the original condition, i.e., x and y. We define the set $V(s)$ as all the variables that are reachable to s, excluding elements in $U(s)$. [1] Based on set $U(s)$ and $V(s)$, we construct the new condition as follows. We firstly randomly pick $N-2$ variables from $V(s)$, forming the set $\hat{V(s)}$. Then, we generate a list $L(s)$, consisting of elements in $\hat{V(s)} \cup U(s)$. Elements in $L(s)$ is shuffled so that their orders are randomized. In list $L(s)$, we identify the indexes of variable x and y, marked as $i_x(s)$ and $i_y(s)$, respectively. Finally, we assign an unique *condition ID* to s, marked as $i(s)$. With the above constructions, we transform the condition of s into a *control flow query* (*CFQ*) function call `cfQuery`, defined as follows:

$$\text{if (} \texttt{cfQuery(L(s), i(s))} \text{) then } \{\text{goto L}\} \tag{3}$$

In the parameter list ($L(s)$) involved in the `cfQuery` function call, the elements selected from $\hat{V(s)}$ are called the *confusing parameters*, the elements selected from $U(s)$, i.e., x and y, are called the *actual parameters*, the value of N are called the *parameter length*.

The control flow matrix M contains logics of all conditions. For each branch statement s, a tuple will be created in M, recording the condition ID, the indexes of the actual parameters appeared in the parameter list $L(s)$, and the operator of the original condition op, marked as

$$< i(s), i_x(s), i_y(s), op > \tag{4}$$

During the execution, the function `cfQuery` will be executed in the CF Enclave. Based on the received parameter $i(s)$, `cfQuery` will identify the corresponding tuple in M. Based on $i_x(s)$ and $i_y(s)$ in the tuple, it will select variable x and y from $L(s)$, restore the condition by combining op, and return the evaluation result of the condition.

In the condition expression in (1), if only one variable, e.g., x, exists and the other is a literal l with a type t, we generate the tuple like $< i(s), i_x(s), string(l), op, string(t) >$, where $string()$ is a string representation of a type or a literal. During the runtime, `cfQuery` will restore the value of l based on the type t and reassemble the condition to evaluate.

2) Fake Branch Statements Insertion: To further obfuscate the control flow, we propose a variant of the basic scheme, namely *fake branch statements insertion*. The idea is to insert some branch statements before the original statements. During the program execution, the inserted branch statement will choose the branch leading to the correct statement, such that the runtime control flow will not be tampered. To distinguish the inserted branch statements from the branch statements transformed through the basic scheme, we call the former as the *fake branch statements* and the latter as the *non-fake branch statements*.

The fake branch statements insertion is performed as follows. For an original statement s, we first collect all the variables that are reachable to s, marked as $V(s)$. Then we randomly pick N variables from $V(s)$, and generate a random-ordered list $L(s)$. Based on $L(s)$, we construct a branch statement $\hat{s}$ and assign a unique ID to $\hat{s}$ as the condition ID, marked as $i(\hat{s})$. Finally, we insert $\hat{s}$ before s. The format of $\hat{s}$ is similar to the basic scheme, as shown below.

$$\text{if (} \texttt{cfQuery(L(s), i(}\hat{s}\texttt{))} \text{) then } \{\text{goto L}\} \tag{5}$$

The condition in $\hat{s}$ will return a fixed boolean value b. If b is $true$, L will be the label of s; if b is false, L will be the label of another statement rather than s. The tuple in M that corresponds to $\hat{s}$ is as follows. It only contains the condition ID and the fixed return value.

$$< i(\hat{s}), b > \tag{6}$$

The CFQ function is compatible with fake branch statements. When $i(\hat{s})$ is passed to `cfQuery`, the function locates the tuple $< i(\hat{s}), b >$ and returns b.

To make a trade-off between the obfuscation effect and the performance, for each original statement, we insert a fake branch statement with a certain probability, called *confusion degree*, marked as d.

Fig. 2 shows a concrete example of program transformation. The transformation process is shown in the left part of the figure. For the original branch statement `if(x>y) goto L2`, its condition is replaced with a function call `cfQuery((a,y,b,x), B1)`, where y and x are the actual parameters, a and b are the confusing parameters, and $B1$ is the condition ID. For other statements in the original program, e.g., the statement with the label $L2$, the transformation inserts a fake branch statement before it. The corresponding tuples in the control flow matrix and a sample implementation of `cfQuery` are shown on the right side of the figure.

[1] We define a reachable variable v to a statement s as the variable that has been declared and initialized when s is to be executed.

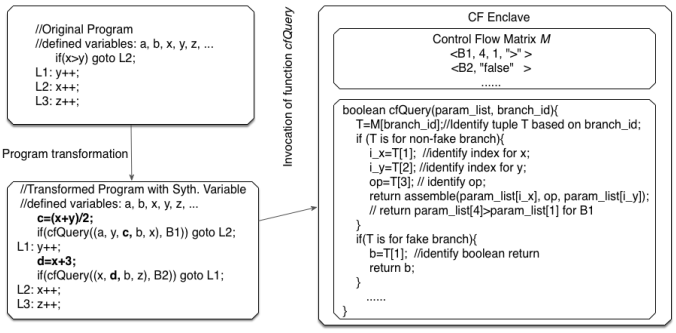


Fig. 2: An example of the program transformation

3) Defeating Passive Attacks: By observing the parameter values of CFQ invokes, the attacker might be able to restore the original conditions or identify fake-branch statements. We call it the *passive attack*. In this section, we first provide concrete examples of this attack. Then we describe scenarios that can prevent this attack. Finally, we discuss solutions in defeating it.

The passive attack on a non-fake branch statement s can be successful in the following scenario. Suppose the CFQ invoke expression in s is f, we mark the parameter list of f as $L(s) = \{p_1, \cdots, p_{i-1}, p_i, p_{i+1}, \cdots, p_N\}$. If the attacker observe that in one invocation of f that returns `true`, the values of parameters satisfy $p_1 < \cdots < p_{i-1} < p_i < p_{i+1} < p_{i+2} < \cdots < p_N$, and in another invocation of f that returns `false`, the values of parameters satisfy $p_1 < \cdots < p_{i-1} < p_{i+1} < p_i < p_{i+2} < \cdots < p_N$, i.e., only the relationship of variable p_i and p_{i+1} is altered, while the relationship of any other two variables are unchanged. The attacker thus can determine that the actual parameters of f are p_i and p_{i+1}. We identify *obfuscation invariant*, which prevents the happening of the above scenario, thus preventing the attacker from identifying the actual parameters.

Theorem 1. *Suppose the CFQ invoke expression f of a non-fake branch statement has a parameter list $L(s) = \{p_1, \cdots, p_N\}$, where p_i and p_j ($1 \le i, j \le N$) are two actual parameters and others are confusing parameters. If the values of $L(s)$ satisfy the following **obfuscation invariant**, the attacker cannot identify p_i and p_j as the actual parameters.*

Obfuscation invariant: there exists a confusing parameter p_c, where in each invocation of f, the value of p_c is always between p_i and p_j, i.e., either $p_i \le p_c \le p_j$ or $p_j \le p_c \le p_i$.

Proof. Our proof strategy is to enumerate all the scenarios of two invocations of f, where in each scenario, the returns of two invocations are different, i.e., one returns `true` and one returns `false`. We prove that, for each scenario where the relationship of two actual parameters is changing, the obfuscation invariant will force the change of other relationships, thus preventing the attacker from identifying the actual parameters.

In each scenario, we mark the value of p_k as p_k^T when f returns `true`, and mark the value of p_k as p_k^F, when f returns `false`. Due to the page limit, we only list the representative scenarios. The other cases are symmetric to the representative ones, and thus are skipped. For instance, the symmetric case of case 1) is $p_i^T < p_c^T < p_j^T$, and $p_i^F > p_j^F$. The scenarios include:

1) $p_i^T > p_c^T > p_j^T$, and $p_i^F < p_j^F$;
2) $p_i^T > p_c^T > p_j^T$, and $p_i^F = p_j^F$;
3) $p_i^T = p_c^T = p_j^T$, and $p_i^F > p_j^F$;
4) $p_i^T > p_c^T = p_j^T$, and $p_i^F < p_j^F$;
5) $p_i^T = p_c^T > p_j^T$, and $p_i^F < p_j^F$;
6) $p_i^T > p_c^T = p_j^T$, and $p_i^F = p_j^F$;
7) $p_i^T = p_c^T > p_j^T$, and $p_i^F = p_j^F$;

The listed cases and their symmetric cases form a complete set of scenarios that satisfy the obfuscation invariant. The proof of each cases is similar. We thus only prove case 1) and skip that of others. The proof is by contradiction: In the two observations of case 1), where the function returns `true` when $p_i^T > p_c^T > p_j^T$ and returns `false` when $p_i^F < p_j^F$, suppose only one variable relationship is changed, i.e., $p_i > p_j$ is changed to $p_i < p_j$, that means the other variable relationships are unchanged, i.e., $p_i^F > p_c^F$ and $p_c^F > p_j^F$ are unchanged. By combining the two inequalities, we have $p_c^F < p_i^F < p_j^F < p_c^F$, leading to a contradiction. Therefore, more than one variable relationships are changing. □

We provide a concrete example of *obfuscation invariant*. Suppose $L(s) = \{x, y, a\}$, where x, y are actual parameters and a is a confusing parameter. If a is within the values of x and y, in one invocation that returns `false`, the parameter values satisfy $x < a < y$, and in another invocation that returns `true`, the parameter values satisfy $x > a > y$. When f's return changes from `false` to `true`, three relationships, i.e., (x, y), (x, a) and (y, a) are all changed. Therefore, the attacker can not identify the actual parameters.

The passive attack on fake branch statement s can be successful in the following scenario. Suppose the CFQ invoke expression in s is f, whose parameter list is $L(s) = \{p_1, \cdots, p_N\}$. If the attack could observe that for each different combination of $p_i, p_j \in L(s)$, and each relationship of p_i and p_j, f will return the same boolean value, the attacker would identify s as a fake branch statement.

We identify the obfuscation invariant that will prevent the attacker from identifying the fake branch statements.

Theorem 2. *Suppose the CFQ invoke expression f of a fake branch statement has a parameter list $L(s) = \{p_1, \cdots, p_N\}$. If the values of $L(s)$ satisfies the following defined **obfuscation invariant**, the attacker cannot identify s as a fake branch statement.*

Obfuscation invariant: there exists two parameters $p_i, p_j \in L(s)$, where in each invocation of f, the mathematical relationship of p_i and p_j is unchanged.

Proof. If the relationship of two parameters, p_i and p_j, is unchanged, the attacker cannot confirm that all six different relationships of p_i and p_j returns the same boolean value. Therefore, she cannot identify s as a fake branch statement. □

```
//Transformed Program with Syth. Variable
//defined variables: a, b, x, y, z, ...
    c=(x+y)/2;
    if(cfQuery((a, y, c, b, x), B1)) goto L2;
L1: y++;
    d=x+3;
    if(cfQuery((x, d, b, z), B2)) goto L1;
L2: x++;
L3: z++;
```

Fig. 3: An example of adding synthetic variables to Fig. 2

Based on Theorem 1 and Theorem 2, our strategy to defeat passive attacks is to ensure that the obfuscation invariants exist for each branch statement. Specifically, we add one or more *synthetic variables* to the parameter list of each CFQ invocation. For non-fake branch statements, the values of those variables are between the values of two actual parameters. For fake branch statements, the value of that variable is always greater than (or always less than) one specified parameter in the parameter list. To achieve that goal, we insert extra statements to compute a value based on existing variables and assign that value to the synthetic variable. There are a variety of ways in generating such variables. we leave an open space for the user to implement their own methods.

Fig. 3 shows an example of adding syntactic variables to the transformed program in Fig. 2. As the figure shows, we add synthetic variable c and d to the two CFQ calls, respectively. Since the value of c is between x and y, according to Theorem 1, passive attack cannot identify the two actual values from x, y and c. Since the value of d is always greater than x, according to Theorem 2, the attacker cannot observe all different relationships between x and d. Thus, the attacker cannot identify the second branch statement as a fake-branch statement.

C. CF Enclave Design

In this section, we introduce the design details of the CF Enclave, including the enclave set up, the LRU cache and the data flow check, which is used to defeat the active attacks.

1) CF Enclave Set Up: The CF Enclave is set up as follows. Once CF Enclave is initiated, a standard SGX remote attestation procedure [15] is performed. During the remote attestation, the measurement of the CF Enclave will be computed and sent to the authentication server. By checking the measurement value against the expected hash value on the authentication server, the authentication server validates the integrity of the CF Enclave. Once the CF Enclave passed the remote attestation, the authentication server will send the decryption key of the control flow matrix, k_M, to the CF Enclave. The decryption key is transmitted in a secure channel generated during the remote attestation. Upon receiving k_M, the CF Enclave will load $E(M)$ and decrypt it with k_M. Finally, the CF Enclave will load M in the enclave's memory, from which the CFQ function can obtain the condition information and offer control flow inquiry service.

2) The LRU Cache: Each invocation of CFQ function is a *ecall*, which takes a longer time than a normal system call (more than 54 times of CPU cycle numbers, according to [16]). We implemented a software-based cache outside of the CF Enclave, called the *LRU cache*, to reduce the frequency of CFQ calls. The LRU cache is implemented as a hash table, which caches the values of the parameters and return of each CFQ invocation. During the program execution, each CFQ call will first look up the LRU cache by the parameter values, and return the cached return value if there is a cache hit. If a cache miss happens, it will invoke the actual CFQ function in the CF Enclave and store values of the function call parameters and return to the LRU for the future lookup. As its name suggests, LRU cache will perform the *least recent used* cache replacement policy.

3) Defeat the Active Attack: The attacker might invoke the same CFQ function repeatedly with different parameter values. By observing the relationship among the return value and the parameter values, the attacker could derive the condition information of the original program. We call such an attack as the *active attack*. In order to defeat it, we introduce *data flow check* in the CF Enclave.

The idea is to statically analyze the original program and derive the mathematical constraints among variables appeared in the parameter list of each branch statement. During the runtime, when the CF Enclave receives a CFQ call, it will check the passed parameter values against the derived constraints. Any inconsistency indicates a violation of the data flow integrity.

The mathematical constraints are derived through a standard intra-procedure forward data flow analysis. The algorithm is shown in Algorithm 1. In the algorithm, $In(s)$ and $Out(s)$ are the variable constraints before and after executing statement s. $Gen(In(s), s)$ returns the new constraints generated after executing s. $Kill(In(s), s)$ returns the constraints that no longer stand (i.e., needs to be removed) after executing s. The algorithm initialize $In(s)$ and $Out(s)$ to $\emptyset$. In each iteration of the while loop, the algorithm updates $In(s)$ based on all the exit states of s's predecessors (line 5); it also updates $Out(s)$ based on $In(s)$ and the semantics of statement s (line 6-10). The loop exits when the $Out(s)$ for each statement s is unchanged. The resulting $In(s)$ for each statement will be returned as a set.

The definitions of function $Gen(In(s), s)$ and $Kill(In(s), s)$ are shown in Table I. In Table I, $\Diamond$ is a binary operator, defined as follows.

$$\Diamond ::= + \mid - \mid * \mid / \mid XOR \mid AND \mid OR \mid \cdots \quad (7)$$

$a \prec r$ means variable a is contained in constraint r. If the executed statement is in the format of $a := b \Diamond c$ (case 1), the generated constraint will be $a = b \Diamond c$ and the killed constraint will be the elements in $In(s)$ that contain a. For statements of format $a := a \Diamond b$ (case 2), we generate the new constraint by replacing a in existing $In(s)$ with $\Diamond^{-1}(a, b)$, i.e., the expression of the old a (the one on the right of $:=$) with the

TABLE I: Specification of $Gen(In(s), s)$ and $Kill(In(s), s)$

case #	Statement s	Gen(In(s),s)	Kill(In(s), s)
1	$a := b \Diamond c$	$\{a = b \Diamond c\}$	$\{r : r \in In(s) \wedge a \prec r\}$
2	$a := a \Diamond b$ or $a := b \Diamond a$	$\{r[a \rightarrow \Diamond^{-1}(a,b)] : r \in In(s) \wedge a \prec r \wedge \exists \Diamond^{-1}(a,b)\}$	$\{r : r \in In(s) \wedge a \prec r\}$
3	if $(x\ op\ y)$ then goto L, true branch	$\{x\ op\ y\}$	$\{\emptyset\}$
4	if $(x\ op\ y)$ then goto L, false branch	$\{\neg\ x\ op\ y\}$	$\{\emptyset\}$
5	$a := func(\cdot)$	$\{\emptyset\}$	$\{r : r \in In(s) \wedge a \prec r\}$
6	others	$\{\emptyset\}$	$\{\emptyset\}$

Algorithm 1 Variable Relation Mining Algorithm

```
1: Out(s) ← {∅} for all statements s
2: W ← {all statements}
3: while W! = {∅} do
4:    Take s from W
5:    In(s) ← ∩_{s'∈pred(s)} Out(s')
6:    temp ← Gen(In(s), s) ∪ (Kill(In(s), s))
7:    if temp! = Out(s) then
8:       Out(s) ← temp
9:       W ← W ∪ succ(s)
10:   end if
11: end while
12: return {In(s)}
```

new a (the one on the left of :=). For instance, suppose the statement is $a = a + b$, and existing constraint is $a > 5$, $\Diamond^{-1}(a, b)$ will be $a - b$, and the constraint is updated to $a - b > 5$. The idea is that existing constraints still stand for the old a, but do not stand for the new a. However, after executing s, the old a will not exist. In order to keep the constraint, we need to replace the old a with an expression using new a.

For a branch statement (case 3 and 4), we simply add the condition as a new constraint. For a function call statement (case 5), we remove all the constraints in $In(s)$ that contain a, since a might be updated in the $func(\cdot)$. For other cases (case 6), we neither generate nor remove any constraint.

When the algorithm terminates, the $In(s)$ of each statement s contains a set of constraints among variables. During the program execution, when the CF Enclave receives a CFQ call on the branch statement s, it will check the concrete parameter values against the constraints in $In(s)$. Any inconsistency indicates a violation of data flow integrity. Upon detecting a violation, the CF Enclave will exit and the execution of the program will be terminated.

V. Security Analysis

In this section, we analyze the security guarantee of CF Hider. We show that the CFHider can defeat static and dynamic, passive and active attacks.

A. On Defeating the Static Analysis

Since the condition flow matrix is hidden in the trusted area, to the attacker, each condition in the transformed program appears to be independent from others. For a CFQ invocation statement whose parameter list length is N, its parameter list can only tell the attacker the variables that are *possibly* involved in the original condition. When the attacker randomly guesses the actual parameters based on the "possible" variables, we are analyzing the probability of restoring the original condition.

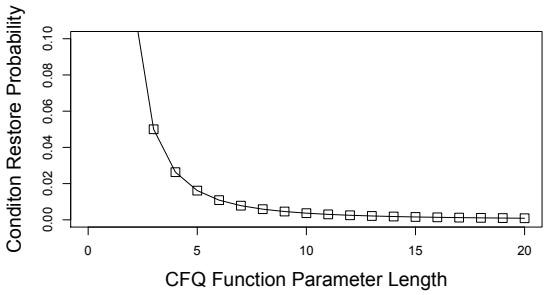


Fig. 4: Condition restore probabilities $R_{branch}(N)$ with different parameter length N.

The condition to be restored can be either a fake branch condition or a non-fake branch condition. The fake branch condition can have two possibilities, i.e., $true$ or $false$. If it is a non-fake branch condition, two out of N parameters are the actual parameters and one out of six operators is the correct operator, resulting in $6 \cdot \binom{N}{2}$ different possibilities. Therefore, to restore a CFQ function, there are in total $6 \cdot \binom{N}{2} + 2 = 3N^2 - 3N + 2$ different possibilities. Since each possibility has equal probability, the probability for the attacker to make a correct guess is $R_{branch}(N) = \frac{1}{3N^2-3N+2}$

Fig.4 simulates the condition restore probability $R_{branch}(N)$ with different parameter length N. As the figure shows, the condition restore probability decreases quickly as CFQ function parameter length N increases. When N exceeds 10, the probability will be close to 0.

B. On Defeating the Passive Attack

If the attacker performs the passive attack described in Section IV-B3, according to Theorem 1 and Theorem 2, the obfuscation invariants can prevent the attacker from identifying the actual parameters or detecting fake branch statements. Since the synthetic variable will be passed to the CFQ function as a parameter, the attacker cannot determine whether it is used in evaluating the branch statement condition, thus cannot determine whether it is synthetic. Although the obfuscation invariant is a characteristic of synthetic variables, we argue that normal variables can also satisfy the obfuscation invariants. Thus it is difficult to identify them by analyzing variable relationships.

TABLE II: Specifications of Hadoop applications

Name	Description
Word Count	Count the number of words from 1 GB of text file. The job contains 20 map tasks, 1 reduce task.
PI	Estimate π using the Monte-Carlo method. The job contains 10 map tasks, where each task generates 100,000 samples.
Tera Sort	Sort 1 GB of text strings in the alphabetical order. The job contains 18 map tasks and 1 reduce task.
Page Rank	Rank 50,000 web pages using PageRank algorithm implemented in *Pegasus*, a Hadoop-based graph mining system. The application contains six jobs. Each job contains 4 to 8 map tasks and 2 reduce tasks.

C. On Defeating the Active Attack

Since the control flow matrix cannot be accessed by the attacker, the attacker cannot derive a valid variable range. If the attacker performs active attack, invoking CFQ functions with arbitrary parameter values might violate the relationship among variables, which will be detected by the data flow checker. Since CF Enclave will exit if it detects only one violation of the data flow integrity, such information is very limited to the attacker to derive the variable relationships among all variables.

VI. Implementation and Experiments

A. Implementation

We implemented a prototype system targeting on protecting the control flow confidentiality of Java programs. We use *Soot*, an open source Java-based compiler tool, to perform program analysis and transformation. Our system can be applied directly to the byte code (i.e., the *class* files) of the program, therefore does not require the source code. The *class* files of the program are analyzed and transformed in *Jimple* code, a typed, three-address, statement-based intermediate representation.

To implement the data flow check, we first implemented the variable relation mining algorithm by employ Soot and *Symja*, a Java-based computer algebra system, which translates statements into mathematical constraints, as specified in Table I. We employ Symja's built-in function `Solve` to generate the inverse operation $\Diamond^{-1}$ of some statements (case 2 in Table I). Inside the CF Enclave, we implemented a simple computer algebra system to validate the passed parameter values against the constraints.

The CF Enclave is implemented with Intel SGX SDK 1.7. The CFQ function is implemented as an `ecall` function.The components in CF Enclave are implemented in C++. The components in the untrusted area are implemented in Java. The two parts are connected with *Java Native Interface* (*JNI*). In JNI, we convert some types in Java into certain C++ types. For example, we convert types *byte*, *short*, *boolean*, and *object* into *int* type. For each *object*, we use its hash code (an integer) in the C++, which supports object operations such as $==$ and $!=$.

B. Experiments

Based on our prototype system, we performed a series of experiments on SGX-supported computers to measure the performance overhead. The experiments were divided into two sets. The first set tested CFHider on Hadoop applications, which helped us to understand the performance overhead when protecting a popular big data framework. The second set tested CFHider on CPU intensive applications, which helped us to obtain a better insight into the overhead. The experimental results indicated that CFHider is applicable to all Java (Hadoop) programs that we tested and incurs an acceptable/reduced performance overhead.

TABLE III: Statistics of Control Flow Matrixes

Application	Matrix Size (Bytes)			# of Generated CFQs		
	d=0	d=30%	d=50%	d=0	d=30%	d=50%
Word Count	36	142	221	3	15	24
PI	188	1.0k	1.7k	18	110	182
Tera Sort	108	668	979	11	73	109
Page Rank	7.4k	31.0k	46.8k	627	3247	4999

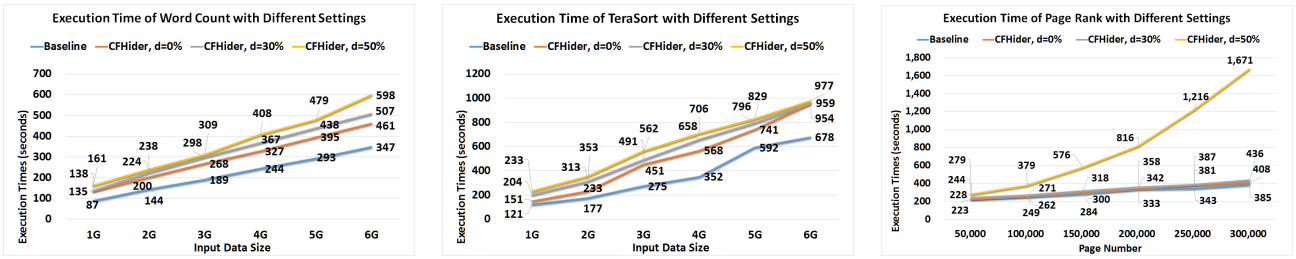
1) Hadoop Applications: In this set of experiments, we set up a cluster consisting of four Lenovo Thinkpad T460 laptops. Each laptop was running Linux Ubuntu 14.04 and equipped with an i7-6500U Intel CPU, supporting SGX Version 1, 8 Gigabytes of memory and 500 Gigabytes of hard disk. All test cases were executed in the SGX hardware release mode, which reflected the production mode performance. In this cluster, one worked as both the master and the worker, and the other three worked as workers. When programs are executed on a cluster consisting of multiple physical hosts, each host will have a dedicated CF Enclave.

We performed the experiments using Hadoop 1.0.4. The tested applications are listed in Table. II, where the first three were selected from the Hadoop 1.0.4 release package, and the last one was selected from the Hadoop benchmark suite HiBench 3.0.0 [17].

The statistics of the control flow matrixes generated during the transformation are listed in Table III. The table indicates that for each application, the matrix size increases with d. On average, each generated CFQ function takes 10 bytes of space in the matrix.

For each application, we transformed all the class files in the application package (i.e., the *jar* file) with the control flow transformer. For each application, we set the parameter length (N) of the CFQ function as 10, the LRU cache size (W) as $10,000$, and the confusion degree (d) as 0%, 30% and 50%, respectively. To compare, we selected the baseline as the corresponding original Hadoop application. The experimental results indicated that CFHider is a general system that can be applied to all the applications that we selected.

The performance results are shown in Fig. 5. Fig. 5 indicates that when setting d as 0%, the average performance overhead of the four applications is 20.75%. The results also indicated that introducing more fake branch statements (i.e., increasing



(a) (b) (c)

Fig. 6: The Scalability Experiments Results. a) Word Count. b) Tera Sort. c) Page Rank.

the confusion degree d) will introduce a higher performance overhead. When d was set to 30%, the average performance overhead of the four applications were 34.06%; when d was set to 50%, the average performance overhead of the four applications were 57.99%.

We selected three applications to perform scalability tests to observe the increase trend of the execution time as the input data size increases. In this set of tests, we still set N as 10, W as $10,000$, and d as 0%, 30%, and 50%, respectively. For word count and tera sort, we increased the input data size from 1 Giga Bytes to 6 Giga Bytes. For the page rank, we increased the number of pages to be processed from 50,000 to 300,000 gradually. To compare, we selected the baseline as the execution time of the original applications under the same input size. The execution time for each setting were recorded in Fig. 6. According to Fig. 6, for most cases (except for the case of page rank with d setting to 50%), the execution time increased almost linearly as the input size increased, and the increase rate of the execution time was similar to that of the baseline. We notice that when the confusion degree d is set to 50%, the execution time of page rank increased faster than the baseline. Our investigation found that the fake branch statements were inserted in frequently executed loop blocks, which incurred a higher number of CFQ invokes. In general, CFHider has shown a good scalability.

2) CPU intensive applications: The overheads of Hadoop applications were amortized since only the customer programs were executed under CFHider, whereas the Hadoop framework and the network communication were unaffected. To better understand the overhead of CFHider, we used a single Lenovo T460 laptop (introduced in Section VI-B1) to study the performance overhead of three CPU intensive Java applications, i.e., binary search, bubble sort and quick sort, and listed the details in Table IV.

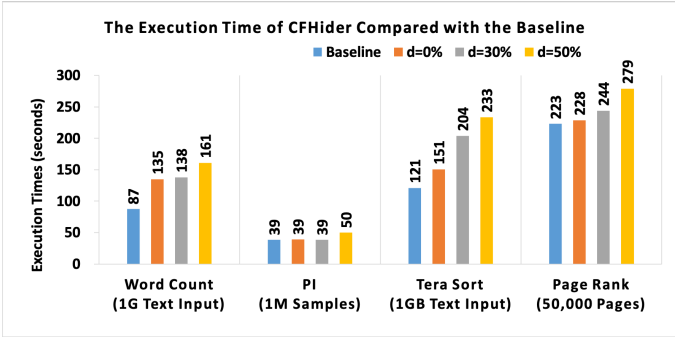


Fig. 5: The execution time of Hadoop applications

In this set of experiments, the data set of each test case was 100 randomly generated integers. When running under CFHider, we set N as 10, set W and d as 0%, meaning disabling LRU cache and not inserting fake branch statements. According to Table IV, different applications incurred different performance overheads ranging between 2.9 times and 3.7 times. We further studied the execution details and found that the overhead is majorly made up of CFQ invocations. A higher CFQ number incurs a higher overhead. The average time for each CFQ function invocation is $4.31\mu s$. Compared to the software-base solution such as [4], where executing each protected condition costs $38.19\mu s$, our solution reduced the overhead by a factor of 8.8.

VII. Related Work

A. Program Confidentiality Protection

Control flow obfuscation [18] has been extensively explored in the past decades. As discussed in Section I, existing software-based methods [1] [3] [4] cannot satisfy security, performance and generality at the same time. For instance, Wang et al. [19] introduced unsolved mathematical conjectures into branch conditions to increase the difficulty of symbolic execution-based reverse engineering. However, this method can be defeated by pattern matching because the number of known conjectures is limited. Lan et al. [4] and Wang et al. [5] proposed to replace the conditional instructions with lambda calculus and Turing machine simulations, respectively, which can defeat symbolic execution-based reverse-engineering attacks. However, their solutions incurs a much higher performance overhead than ours. Wang et al. [20] worked on the hybrid cloud setting and proposed to move the conditions to

TABLE IV: Execution details of CPU intensive programs

Application	Execution time		Overhead	CFQ Invoke #	CFQ Time
	Original	CFHider			
Binary Search	$41ms$	$161ms$	2.9	121	$4.46\mu s$
Bubble Sort	$48ms$	$227ms$	3.7	10,200	$4.26\mu s$
Quick Sort	$40ms$	$188ms$	3.7	2,292	$4.22\mu s$

the private cloud, while leaving other computation on the public cloud. By leveraging the proposed *continuous cache*, their solution achieves a moderate performance overhead (14.9% to 33.2%), however cannot provide a strict security guarantee.

B. SGX Applications

Intel Software Guard Extension (SGX) [15] is receiving wide attentions recently. A number of systems have been proposed to offer secure computing services for different application settings. Haven [6] runs legacy windows applications on commodity OS and SGX-enabled hardware, improving the computing integrity and confidentiality. However, Haven requires a library OS variant loaded into an enclave, making the TCB size much larger, therefore cannot offer a guaranteed security, but only raising the bar. Graphene-SGX [7] ports Graphene Library OS into SGX, enabling a set of Linux applications to be executed on SGX enclave, such as Lighttpd, Apache, NGINX. Similar to Have, this work only reduced the TCB size, yet still does not provide any security guarantee. Glamdring [9] and Ryoan [21] focused on protecting sensitive data confidentiality, but can not protect program control flow confidentiality. Some works leverage SGX to protect execution security for a specific application, including database system (EnclaveDB) [22], C++ version of MapReduce (VC3) [10], Apache ZooKeeper [23], etc.

VIII. Conclusion

In this paper, we proposed and implemented a hardware-assisted solution, namely CFHider, to protect the control flow confidentiality in the public cloud setting. Our theoretical analysis and experiments indicate that CFHider is effective to protect the control flow confidentiality, satisfying the requirements of security, expressiveness and performance.

IX. Acknowledgements

The authors thank the anonymous reviewers for their valuable feedback. The corresponding author is Yulong Shen. This research is supported in part by the National Natural Science Foundation of China (NSFC) (No. 61602364), Natural Science Foundation of Shaanxi Province (No. 2017JM6083), NSFC Grants U1536202, U1736216 and 61571352, Shaanxi Science & Technology Coordination & Innovation Project 2016KTZDGY05-07-01 and National Key R&D Program of China (No. 2017YFB1400700).

References

[1] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*. IEEE, 2001, pp. 193–202.

[2] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: Reverse engineering obfuscated code," in *Reverse Engineering, 12th Working Conference on*. IEEE, 2005, pp. 10–pp.

[3] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation." in *NDSS*, 2008.

[4] P. Lan, P. Wang, S. Wang, and D. Wu, "Lambda obfuscation," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2017, pp. 206–224.

[5] Y. Wang, S. Wang, P. Wang, and D. Wu, "Turing obfuscation," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2017, pp. 225–244.

[6] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 267–283.

[7] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[8] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. Stillwell *et al.*, "SCONE: Secure linux containers with Intel SGX," in *OSDI*, vol. 16, 2016, pp. 689–703.

[9] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza *et al.*, "Glamdring: Automatic application partitioning for Intel SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 285–298.

[10] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 38–54.

[11] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13, 2013.

[12] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." in *HASP@ ISCA*, 2013, p. 10.

[13] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 991–1008.

[14] V. Aho Alfred, S. Ravi, and D. Ullman Jeffrey, "Compilers: principles, techniques, and tools," *Reading: Addison Wesley Publishing Company*, p. 466, 1986.

[15] V. Costan and S. Devadas, "Intel SGX explained." *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[16] O. Weisse, V. Bertacco, and T. Austin, "Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 81–93.

[17] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, March 2010, pp. 41–51.

[18] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1998, pp. 184–196.

[19] Z. Wang, J. Ming, C. Jia, and D. Gao, "Linear obfuscation to combat symbolic execution," in *European Symposium on Research in Computer Security*. Springer, 2011, pp. 210–226.

[20] Y. Wang and J. Wei, "Toward protecting control flow confidentiality in cloud-based computation," *Computers & Security*, vol. 52, pp. 106–127, 2015.

[21] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: a distributed sandbox for untrusted computation on secret data," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016, pp. 533–549.

[22] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A secure database using SGX," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 405–419.

[23] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper using Intel SGX," in *Proceedings of the 16th Annual Middleware Conference (Middleware)*, 2016.