Control Flow Obfuscation for FJ using Continuation Passing
Extended Version

Kenny Zhuo Ming Lu
School of Information Technology
Nanyang Polytechnic
luzhuom@gmail.com
Information Systems Technology and Design
Singapore University of Technology and Design
kenny_lu@sutd.edu.sg

Abstract
Control flow obfuscation deters software reverse engineering attempts by altering the program’s control flow transfer. The alternation should not affect the software’s run-time behaviour. In this paper, we propose a control flow obfuscation approach for FJ with exception handling. The approach is based on a source to source transformation using continuation passing style (CPS). We argue that the proposed CPS transformation causes malicious attacks using context insensitive static analysis and context sensitive analysis with fixed call string to lose precision.

CCS Concepts · Software and its engineering → Semantics; · Security and privacy → Software and application security.

Keywords Control flow obfuscation, program transformation, continuation passing style

ACM Reference Format:
Kenny Zhuo Ming Lu. 2020. Control Flow Obfuscation for FJ using Continuation Passing: Extended Version. In Proceedings of ACM Conference (Conference’17). ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnn.nnnnnn

1 Introduction
Java applications are ubiquitous thanks to the wide adoption of android devices. Since Java byte-codes are close to their source codes, it is easy to decompile Java byte-codes back to source codes with tools. For example, javap shipped with JVM [14] can be used to decompile Java class files back to Java source. This makes the Man-At-The-End attack as one of the major security threat to Java applications. Code obfuscation is one of the effective mechanism to deter malicious attack through decompilation. There are many obfuscation techniques operating on the level of byte-codes [5, 16, 19]. In the domain of source code obfuscation, we find solutions such as [8] applying layout obfuscation. We put our interest in control flow obfuscation techniques, which include control flow flattening [3, 10, 20] and continuation passing [11]. Note that the difference between bytecode obfuscation and source code obfuscation is insignificant, because of the strong correlation between the Java bytecodes and source codes. In this paper, we propose an extension to the continuation passing approach to obfuscate FJ with exception handling.

We assume the attackers gain access to the byte-codes to which layout obfuscation has been applied. The attackers decompile the byte-codes into source codes and attempt to extract secret information by running control flow analysis on the decompiled code. Our goal here is to cause the control flow analysis become imprecise or more costly in computation.

2 Motivating Example

Example 1. To motivate the main idea, let’s consider the following Java code snippet

```java
class FibGen {
    int f1, f2, lpos;
    FibGen () {
        f1 = 0; f2 = 1; lpos = 1;
    }
    int get(int x) {
        int i = lpos;
        int r = -1;
        try {
            if (x < i) {
                throw new Exception();
            }
        } else {
            while (i < x) {
                int t = f1 + f2;
                f1 = f2; f2 = t; i++;
            }
        }
    }
}
```
we catch the exception and print out the error message.

In the above we define a Fibonacci number generator in class FibGen. In the method get, we compute the Fibonacci number given the position as the input. Note that generator maintains a state, in which we record the last two computed Fibonacci numbers, namely, f1 and f2 and the last computed position lpos. In method get lines 10 and 11, we raise an exception if the given input is smaller than i which has been initialized to lpos. Towards the end of the method, we catch the exception and print out the error message.

The number comment on the right of each statement indicates the code block to which the statement belongs. In Figure 1, we represent the function get’s control flow as a graph. Each circle denotes a code block from the source program.

Inspired by the approach [11], our main idea is to translate control flow constructs, such as sequence, if-else, loop into CPS combinators. In the context of FJ with exception handling, we translate try-catch statement into a CPS combinator as well.

In Figure 2 we find the obfuscated code snippet of get method in CPS style. The obfuscated code is in a variant of FJ, named FJ, which is FJ with higher order functions, nested function declaration and mutable variables in function closures. void => void denotes a function type whose values accept no argument and return no result. Exception => void denotes a function type that accepts an exception and return no result. type NmCont = void => void defines a type alias. (void n) => {return i < x}; defines an anonymous function whose input is of type void and the body returns a boolean value. For brevity, we omit the type annotations of the formal arguments where there is no confusion. The return key word is omitted when there is only one statement in the function body. We omit curly brackets in curried expressions, e.g. x -> raise -> k -> { ... } is the same as x -> { raise -> { k -> { ... } } }, where x, raise and k are formal arguments for the lambda abstractions. For convenience, we treat method declaration and lambda declaration as interchangeable. For instance, the lambda declaration

\[
\text{int } \Rightarrow \text{int } \Rightarrow \text{int } f = x \Rightarrow y \Rightarrow \{ x + y \}
\]

is equivalent to the following method declaration

\[
\text{int } f (\text{int } x, \text{int } y) \{ \text{return } x + y; \}
\]

In the last section, we mentioned that the layout obfuscation such as identifier renaming should have been applied to the obfuscated code; however in this paper we keep all the identifiers in the obfuscated code unchanged for the ease of reasoning. For the sake of assessing the obfuscation potency, we “flatten” the nested function calls into sequences of assignment statements. For example, let x and y be variables of type int, let f be a function of type int => int => int and g be a function of type int => int; instead of

\[
\text{int } r = f(x)(g(y));
\]

we write:

\[
\text{int } r = f_x = f(x);
\]

\[
\text{int } g_y = g(y);
\]

\[
\text{int } r = f_x(g_y);
\]

As we observe in Figure 2, all the building blocks are continuation functions with type CpsFunc The simple code blocks (1), (6), (7) and (8) from the original source code, which contain no control flow branching statements, are translated into nested CPS functions get1, get6, get7 and get8. Block (4) containing a throw statement is translated into get4 which applies the exception object to the exception handling continuation raise of type ExCont. Block (9) has a return statement, which is translated into a function in which we assign the variable being returned r to the res variable and call the normal continuation k of type NmCont. Block (2) is a try catch statement which is encoded as a call to the trycatch combinator in line 24. Similarly block (3) the if-else statement is encoded as a call to the ifelse combinator and block (5) the while loop is encoded as a call to the loop combinator.

In Figure 3, we present the definitions of the CPS combinators used in the obfuscation. Combinator loop accepts a condition test cond, a continuation executor visitor to be executed when the condition is satisfied, a continuation executor exit to be activated when the condition is not satisfied. Combinator seq takes two continuation executors and executes them in sequence. Combinator trycatch takes a continuation executor tr and an exception handling continuation hdl. It executes tr by replacing the current exception continuation with ex_hdl. Combinator ifelse accepts a condition test cond, a continuation for the then-branch th
to be executed when the condition is satisfied, a continuation executor for the else-branch el to be activated when the condition is not satisfied.

To assess the potency of the obfuscation technique, let’s put on the hat of the attackers and apply some static analysis to the obfuscated source code. The goal of the attack is to reconstruct the control flow graph from the obfuscated source. We apply an inter-procedural data flow analysis to the obfuscated code. For each variable or formal argument in the code, the analysis tries to approximate the set of possible lambda expressions which the variable/argument may capture during the execution. From the approximation we re-create the (global) control flow graph as presented in Figure 4. We give names to anonymous functions as λ_l where l refers to the line number appearing in Figures 2 and 3.
We show that CPS based control flow obfuscation is effective against static analysis, in particular context insensitive control flow analysis.

The rest of the paper is organized as follows. In Section 3, we formalize SSAFJ-EH’s syntax and semantics. In Section 4, we define the syntax of FJ as well as its semantics. We formalize the source-to-source translation from SSAFJ-EH to FJ. In Section 5, we discuss in details about the potency assessment of our obfuscation technique against static analyses. We discuss about related works in Section 6 and conclude in Section 7.

3 Single Static Assignment Form for FJ with Exception Handling

3.1 Syntax of SSAFJ-EH

We extend the syntax of SSAFJ [1] with exception handling,

```
(classDecl) cd ::= class C (fd, md)
(fieldDecl) fd ::= t f
(methodDecl) md ::= t m (t x) (vd, b)
(varDecl) vd ::= t x
(block) b ::= l : {s}
(statement) s ::= \pi | return e | throw e | x = e.m(e)
                   | try {b} join \phi catch (t x) {b} join \phi | join \phi while e {b}
                   | if e {b} else {b} join \phi
(assignment) a ::= x = e | e.f = e
(phi) \phi ::= x = phi(T : x)
(label) l ::= L_0 \mid L_1 \mid L_2 \mid ...
(expression) e ::= v | x | e.f | new t() | this | e op e
(operator) op ::= + | - | [=] | == | ...
(type) t ::= int | bool | void | C
(value) v ::= c | loc | null
(memLoc) loc ::= loc(0) | loc(1) | ...
```

class C {fd; md} defines a class. C denotes a class name. fd denotes a sequence of field declarations, fd_1; ...; fd_n. Likewise for md denotes a sequence of method declarations. For simplicity, we do not consider class inheritance, class constructors and method modifiers. Implicitly we assume each class comes with a default constructor and all field declarations are public and non-static. t m (t x) {vd, b} defines a method declaration. For simplicity, we restrict the language to single argument methods. m denotes a method name. x, y and z denote variables. b defines a sequence of blocks. Each block is associated with a label l. Labels are unique within the method body. Reference to labels is restricted to the method’s local scope. Each block consists of a sequence of assignment statements or a control flow statement. The last block in a method must contain a return statement. Note that all control flow statements potentially alter the default top-down execution order. The SSA form ensures the definition of a variable through assignment must dominate all the uses of this variable. Unlike the work [11], which uses low-level SSA structure with goto statements, the SSA form introduced in this paper is in a high-level structured form. That is, only certain control flow statements, such as if-else,
try-catch block, while, may carry one or more \( \phi \) clauses. There is no goto statement. A \( \phi \) assignment \( x = \text{phi}(I : x) \) selects the right labeled argument \( I : x \) to assign to the left hand side variable, based on the label of the preceding statement. For if-else statement, the \( \phi \) assignment is inserted right after the then- and else-branches, which merges the possible different set of values from the branches into a new set of variables. In the while loop, the \( \phi \) assignment is located before the loop-condition. In try-catch statement, we find two sets of \( \phi \) assignments. The \( \phi \) assignments located after the try block and catch block have a functionality similar to the one in if-else statement. The other one is located between the try block and the catch block. It is to merge the different sets of values that are arising in various parts of the try block due to exception being raised. We will discuss more in details in the semantics of SSAFJ-EH. \( t \) denotes a type. A type \( t \) can be basic types such as \texttt{int}, \texttt{void} or a class type \( C \). A value \( v \) is either a constant, a memory location or null. The formal details will be elaborated in the upcoming subsection. Syntax of assignments and expressions is standard. For instance, the corresponding SSA form of the method get from the class \texttt{FibGen} in Example 1 is in Figure 5.

### 3.2 Semantics of SSAFJ-EH

We report a call-by-value semantics of SSAFJ-EH in Figures 6 and 7. We adopt the standard denotational semantics notation found in \[15\]. \texttt{GEnv} denotes a constant global environment which maps class names to field declarations and class

![Figure 5](https://example.com/figure5.png)

**Figure 5.** Method get in Single Static Assignment Form

| (Global Decl Env) | \( \text{GEnv} \subseteq (\text{Class} \times \text{FieldDecl}) \cup ((\text{Class} \times \text{Name}) \times \text{FieldDecl}) \) |
| (Local Decl Env) | \( \text{LEnv} \subseteq (\text{Variable} \times \text{Value}) \) |
| (Memory Store) | \( \text{Store} \subseteq (\text{MemLoc} \times \text{Object}) \) |
| (Object) | \( \text{obj} \) \( \equiv \) \( \text{obj}(t, \rho) \) |
| (Exception) | \( \text{ex} \) \( \equiv \) \( \text{exception}(n, \text{LEnv}, \text{Store}, I) \) |
| (Object Field Map) | \( \rho \) \( \equiv \) \( \text{FieldName} \times \text{Value} \) |

| \( \text{MD}_{\text{ssa}}[\cdot] := \text{MethodDecl} \rightarrow \text{Value} \rightarrow \text{Value} \rightarrow \text{GEnv} \rightarrow \text{Store} \rightarrow (Value, \text{Store}, \text{Label})_\text{ex} \) |
| \( \text{MD}_{\text{ssa}}[\cdot, \cdot] := \text{Block} \rightarrow \text{Label} \rightarrow \text{GEnv} \rightarrow \text{LEnv} \rightarrow \text{Store} \rightarrow (Value, \text{LEnv}, \text{Store}, \text{Label})_\text{ex} \) |
| \( \text{B}_{\text{ssa}}[\cdot] := \text{Block} \rightarrow \text{Label} \rightarrow \text{GEnv} \rightarrow \text{LEnv} \rightarrow \text{Store} \rightarrow (Value, \text{LEnv}, \text{Store}, \text{Label})_\text{ex} \) |

![Figure 6](https://example.com/figure6.png)

**Figure 6.** Denotational Semantics of SSAFJ-EH (Part 1)
names and method names to method declarations. We assume that the given program is free of type errors and there is no null pointer reference error. LEnv denotes a local variable environment which maps variables to values. Store defines a memory environment that maps memory locations to objects.

As a convention, we write $m(a)$ to refer to the object $b$ associated with the key $a$ in a mapping $m$, i.e. $(a, b) \in m$, given that all keys in $m$ are unique. We use $m + (a, b)$ to denote an “update if exists - insert otherwise” operation, i.e. $m + (a, b) = \{(x, y) \in m|a \neq x\} \cup \{(a, b)\}$.

In this paper, we are only interested in the obfuscation of methods, hence we omit the semantics for class declaration and field declaration. $\text{MD}_{\text{ssa}}[\cdot]$ defines the semantics of a method as a function expecting a reference to the current object, a value as the actual argument, a global environment and a memory store and returns a pair of value and memory store as result. Given a domain $D$, we write $D_{\text{ex}}$ to denote $D \cup \text{Exception}$. $\text{VDD}_{\text{ssa}}[\cdot]$ takes a list of variable declarations and a local declaration environment as inputs then registers each variable in the declaration environment. Note that we use Haskell’s style of let-binding to introduce temporary variables and case expression for pattern matching. For brevity we omitted data constructors in the patterns when there is no ambiguity.

We adopt Haskell’s style list syntax. $\emptyset$ denotes an empty list. $x : xs$ denotes a non-empty list where $x$ refers to the head and $xs$ refers to the tail. We assume there exists an implicit conversion from a sequence $b_1; b_2; \ldots; b_n$ to a list $b_1 : b_2 : \ldots : b_n : \emptyset$.

$\text{B}_{\text{ssa}}[\cdot]$ evaluates a block with respect to the context, i.e. the label of the preceding block, the local environment and the memory store. As the output, it returns a tuple of four items, namely, the value of the evaluation, the updated local environment, the updated memory store and the label from the exiting block if there is no exception occurred, otherwise an exception is returned. $\text{B}_{\text{ssa}}[\cdot]$ evaluates a sequence of blocks by applying $\text{B}_{\text{ssa}}[\cdot]$ to each block in order, and propagates the resulting environments if there is no exception, otherwise the exception is propagated.

We highlight the a few interesting cases of $\text{B}_{\text{ssa}}[\cdot]$. In case of if-else statement, we evaluate the then-branch $b_1$ or the else-branch $b_2$ depending on the result of the condition expression $e$. Given the label of the exiting block, either from $b_1$ or $b_2$, we apply $\text{B}_{\text{ssa}}[\cdot]$ to update the local environment in the result. In case of a try-catch statement, we first evaluate the try block. If the evaluation is successful, we compute the result by updating the local environment with $\text{B}_{\text{ssa}}[\cdot]$. If some exception arises from the evaluation of the try block, we generate a local environment with $\text{B}_{\text{ssa}}[\cdot]$ depending on the location from which the exception is raised. Next we evaluate the catch block under this local environment. Finally we update the local environment with $\text{B}_{\text{ssa}}[\cdot]$. In case of a method invocation, we evaluate the object expression into a memory location, from which we look up the memory store to retrieve the actual object and its type. From the global environment, we retrieve the method declaration based on the method name $m$. We call $\text{MD}_{\text{ssa}}[md]$ with the actual arguments to compute the result of the right hand side. Finally, we return a tuple consists of a null value, a updated local environment with the updated binding of the left hand side $x$ as well as the updated memory store. The remaining cases are trivial.

$\text{F}_{\text{ssa}}[\cdot]$ walks through the list of $\phi$ assignments. For each $\phi$ of shape $x = \text{phi}(l_1 : x_1, \ldots, l_i : x_i, \ldots, l_n : x_n)$, it searches for the label matching with the incoming label $l_i$. The value of $x_i$ will be assigned to the variable $x$.

The definitions of $\text{F}_{\text{ssa}}[\cdot], \text{A}_{\text{ssa}}[\cdot]$ and $\text{E}_{\text{ssa}}[\cdot]$ are straightforward and we omit the details.
4 SSAFJ-EH to FJλ Translation

4.1 Syntax of FJλ

We consider the valid syntax of our target language FJλ whose value is initialized to a lambda abstraction VD found in the SSAFJ-EH.

The examples given in Section 2.2 differ from SSAFJ-EH.

4.2 Semantics of FJλ

In Figure 8 we describe the denotational semantics of FJλ. We use the upper case symbols GENV, LENV and STORE to capture the run-time bindings. They are similar to the counter-parts found in the SSAFJ-EH.

Figure 8. Denotational Semantics of FJλ

we proceed with the evaluation the body of the lambda abstraction under the new environment and memory store. (II) In case of function application Erjλ[E1,M(E2)], we evaluate E1 into a memory location loc(n) with an updated memory store st’. By looking up st’(loc(n)) we retrieve the definition of the method associated with name M. We then evaluate E2 and apply the resulting value to the method.

4.3 SSAFJ-EH to FJλ Translation using CPS

We describe the SSAF-J-EH to FJλ translation using CPS in Figures 9 and 10. Specifically, we use command-based continuation pass style.

There are mainly two types of continuations, the exception continuation Exception ⇒ void and the normal continuation void ⇒ void. Each function in CPS form expects the first argument as the exception continuation and the second one as the normal continuation, except for top level method.
continuation \((\cdot) \rightarrow k(\mathit{res})\). At last we synthesize the public interfacing method \(M\) which wraps around the CPS counterpart \(M_{\text{cps}}\).\(^2\)

Most of the translation tasks are computed in the helper function \(\text{CB}_{\text{cps}}[\cdot]\). The function expects a list of blocks, a list of \(\phi\) assignments from the subsequent block in the normal continuation and a list of \(\phi\) assignments from the subsequent block in the exception continuation. \(\text{CB}_{\text{cps}}[\cdot]\) translates the blocks structurally.

- In case of a singleton list containing an if-else block, we apply a helper function \(\text{CB}_{\text{cps}}[\cdot]\) to translate the conditional expression. Then we apply \(\text{CB}_{\text{cps}}[\cdot]\) recursively to the blocks from the then-branch and the else-branch by using the \(\phi\) assignments, \(\bar{\phi}\), from the if-else statement’s join clause. To “connect” the translated if-else back to the subsequent block in the normal continuation, we apply another helper function \(\text{CK}_{\text{cps}}[\cdot]\) to construct a continuation that resolves \(\phi_l\) with respect to \(l\). The main expression is constructed structurally from the derived expressions from the various sub-steps with \(\text{seq}\) and \(\text{if} \text{else}\) combinators, whose definitions can be found in Figure 3.

- In case of a non singleton list of which the head is an if-else block, we perform a trick similar to the previous case, except that we do not construct a continuation with \(\text{CK}_{\text{cps}}[\cdot]\) to resolve \(\phi_l\). Instead we apply \(\text{CB}_{\text{cps}}[\cdot]\) to \(\bar{\phi}\) recursively.

- In case of a singleton list containing a while block, we first need to apply \(\text{CK}_{\text{cps}}[\cdot]\) to resolve the \(\phi\) with respect to the label of the block from which we enter the while loop. For convenience, we assume that there exists a partial order among labels, i.e. \(L_i < L_j\) implies that \(L_i\) must be on the path leading from \(L_0\) to \(L_j\), where \(L_0\) is the method’s entry label. We assume that there are only two labels in the \(\bar{\phi}\) assignments in all while blocks, i.e. the first label is the entry label to the while block, and the second label is loop-back label, and \(\text{minLabel}(\bar{\phi})\) returns the entry label. Such a restrictive form does not limit the expressiveness of the language. We assume that there exists a pre-processing step that convert any programs into this form.

After resolving \(\bar{\phi}\) with respect to the entry label to the while block, we apply \(\text{CB}_{\text{cps}}[\cdot]\) recursively to \(\bar{\phi}\) to translate the while body. Lastly we apply \(\text{CK}_{\text{cps}}[\cdot]\) to construct a continuation that resolves \(\phi_l\) with respect to \(l\). We build the main expression using the \(\text{seq}\) and \(\text{loop}\) combinators, whose definitions can be found in Figure 3.

- In case of a singleton list containing a try-catch block, we apply \(\text{CB}_{\text{cps}}[\cdot]\) recursively to the block in the try clause \(\bar{\phi}\) with \(\mathit{phi}_l\) as \(\phi\) assignments from the normal continuation and \(\mathit{phi}_l\) from the exception continuation.

The catch clause block \(\bar{\phi}\) is translated with \(\mathit{phi}_l\) as \(\phi\) assignments from the normal continuation and \(\mathit{phi}_l\) from the exception continuation. In order to bind the exception into the variable \(x\), we define a wrapper lambda expression which expects the exception as input and assigns it to \(ex\). (Recall that \(ex\) is defined in the top level method). Lastly, we construct a connecting continuation by resolving \(\mathit{phi}_k\) with the current label \(l\).

- In case of a singleton list containing a throw block, we first resolve the \(\mathit{phi}_e\) from the exception handler with respect to the current label \(l\) by calling \(\text{CP}_{\text{cps}}[\cdot]\). Taking the result from the \(\mathit{phi}\) resolution, we define a continuation function \(m_l\) in which we bind the results, and call the exception continuation \(\text{raise}(\cdot)\) with the translation of the \(e\).

- In case of a singleton list containing a return block, we define a continuation function \(m_l\) in which we assign the translation of \(e\) to \(\mathit{res}\). (Recall that \(\mathit{res}\) is defined in the top level method). Then we call the continuation \(k\).

- In cases of a list with a method invocation as the head, we translate the sub-expressions \(e_1\) and \(e_2\) into \(E_1\) and \(E_2\). We define a continuation function \(m_l\) in which we invoke \(E_1, m_{\text{cps}}(E_2)\) with \(\text{raise}\) as the exception continuation and the normal continuation is a lambda expression that captures the result of the method invocation into an argument \(v\). In the body of the lambda expression we assign \(v\) to \(x\) before invoking the continuation \(k\). Note that we treat \(M_{\text{cps}}\) same as \(m_{\text{cps}}\) and the call of \(m_{\text{cps}}\) could be a recursive call or another method sharing the same closure context in the same scope.

The rest of the \(\text{CB}_{\text{cps}}[\cdot]\) cases are trivial.

The helper function \(\text{CF}_{\text{cps}}[\cdot]\) takes a list of \(\phi\) assignments, a label and returns a list pair variable-expression pairs. For each \(\phi\) assignment, it picks the right \(x_i\) associated with the matching label \(l\) as the second component of the resulting pair.

The helper function \(\text{CK}_{\text{cps}}[\cdot]\) synthesizes a continuation function that connects the block with label \(l\) with the block that \(\bar{\phi}\) is defined, by making use of \(\text{CF}_{\text{cps}}[\cdot]\).

Helper functions \(\text{CA}_{\text{cps}}[\cdot]\) and \(\text{CB}_{\text{cps}}[\cdot]\) are identity functions, whose definitions are omitted.

In Figure 11, we find the full CPS translation of get function from the F1bGen class. The result should be identical to the one in Figure 2, except that we do not apply “flattenino” to nested and curry function calls, we insert extra connection blocks thanks to the \(\phi\) resolutions.

**Definition 1** (Consistent Global Environments). Let \(\mathit{geno} \in \mathit{GEN}\) and \(\mathit{geno'} \in \mathit{GEN}\). Then we say \(\mathit{geno} \vdash \mathit{geno'}\) iff \(\mathcal{N}(C, m) \in \mathit{dom}(\mathit{geno}) : \mathit{geno'}(C, m) = \text{MD}_{\text{cps}}[\langle \mathit{geno}(C, m) \rangle]\).

**Lemma 4.1** (SSAFJ-EH to FJ1). (Translation Consistency). Let \(m\) be a SSAFJ-EH method of a class \(C\), \(a\) be (a reference to)
int get(int x) {
    int i_1, i_2, i_5, i_6, t_6, r_1, r_2, r_7;
    int input, res; Exception ex;

    int => ExCont => (int => void) => void get_cps =
    x -> raise => k -> {
        input = x;
        return seq(get1), seq(trycatch
            seq({ifelse (n->) => input < i_1),
                get4, seq(getk3b, loop (-=>n->i_5<input),
                  seq(get6, getk6k), seq(get7, get7k))))
            , getk3a)
    , e => {ex = e; return seq(get8, get8k);});
    get9)
    ) (raise) (n->k(res))
    )
    ExCont => NmCont => void get1 =
    (ExCont raise) => (NmCont k) => {
        i_1 = this.iPos; r_1 = -1; return k();
    }
    ExCont => NmCont => void getk3a = raise => k
    -> {r_2 = r_7; return k();}
    ExCont => NmCont => void get4 = raise => k
    -> {i_2 = i_1; raise(new Exception());}
    ExCont => NmCont => void getk3b = raise => k
    -> {i_5 = i_1; return k();}
    ExCont => NmCont => void get6 = raise => k
    -> {t_6 = this.f1 + this.f2; this.f1 = this.f2;
        this.f2 = t_6; i_6 = i_5 + 1; return k();
    }
    ExCont => NmCont => void get6k = raise => k
    -> {i_5 = i_6; return k();}
    ExCont => NmCont => void get7 = raise => k
    -> {this.iPos = i_5; r_7 = this.f2; return k();
    }
    ExCont => NmCont => void get7k = raise => k
    -> {i_2 = i_5; return k();
    }
    ExCont => NmCont => void get8 = raise => k
    -> {System.out.println("..."); return k();
    }
    ExCont => NmCont => void get8k = raise => k
    -> {r_2 = r_1; return k();
    }
    ExCont => NmCont => void get9 = raise => k
    -> {res = r_2; return k();
    }
    get_c(x)(id_raise)(i => res = i; return); return res;
}

Figure 11. SSA to CPS Translation of fib

Let’s try to apply inter procedural control flow analysis
the obfuscated code in Figures 2 and 3. Recall that the
goal of the control flow analysis is to approximate the set of
possible lambda abstractions that a program variable may
capture during the run-time. From that result, as an attacker,
we can create a global control flow graph with all the lamb-
das and methods involved.

Let define the set of all possible lambda values in the
obfuscated program in FJ. We have the following lattice
(2^α, ⊆), whose top element T is Λ and ⊥ is the empty set.
We define the abstract state of the analysis as a map lattice
mapping variables to sets of lambda functions.

\[
\text{(STATE)} \quad \sigma \subseteq (\text{VARIABLE} \times 2^\alpha)
\]

### 5.1 Context Insensitive Control Flow Analysis

We define the flow function \( [](\cdot) : \text{STATE} \rightarrow \text{STATE} \rightarrow \text{STATE} \).
The flow function takes a statement and a state and returns
an updated state. Given a statement \( S \), we write \( [S] \)
to denote \( [S](\sigma) \) by making \( \sigma \) an implicit argument
where

\[
\sigma_S = \text{join}(S)
\]

Let \( S \) be a statement and \( pred(S) \) denote the set of
preceding statements of \( S \), we define the join function
\( \text{join}(S) \) as

\[
\text{join}(S) = \bigcup_{P \in pred(S)} [P]
\]

The definition of flow function is given as follows.

\[
\begin{align*}
\\llbracket \text{return} \rrbracket & (\sigma_S) = \sigma_S \\
\\llbracket \text{if} & \ S \ \text{then} \ S' \ \text{else} \ S'' \rrbracket (\sigma_S) = \sigma_S \\
\\llbracket E_1, F = E_2 \rrbracket (\sigma_S) = \sigma_S \\
\\llbracket X = c \rrbracket (\sigma_S) = \sigma_S - X \cup \{ X \mapsto \emptyset \} \\
\\llbracket X = E.F \rrbracket (\sigma_S) = \sigma_S - X \cup \{ X \mapsto \emptyset \} \\
\\llbracket X = E \ op \ E' \rrbracket (\sigma_S) = \sigma_S - X \cup \{ X \mapsto \emptyset \} \\
\\llbracket X = \text{new} T() \rrbracket (\sigma_S) = \sigma_S - X \cup \{ X \mapsto \emptyset \} \\
\\llbracket X = \lambda \rrbracket (\sigma_S) = \sigma_S - X \cup \{ X \mapsto \{ \lambda \} \} \\
\\llbracket X = Y \rrbracket (\sigma_S) = \sigma_S - X \cup \{ X \mapsto \text{returned} \} \\
\\llbracket X = G(E_1, ..., E_n) \rrbracket (\sigma_S) = \sigma_S - X \cup \{ X \mapsto \text{returned} \}
\end{align*}
\]

where

\[
\text{returned} = \bigcup_{\lambda \in \text{GEN}(G)} \text{returnStmt} \llbracket \lambda \rrbracket
\]

\[
\text{where}
\begin{align*}
\text{returned} & = \bigcup_{\lambda \in \text{GEN}(T,M)} \text{returnStmt} \llbracket \lambda \rrbracket
\end{align*}
\]

The first three cases handle return statement, if statement
and field update. They do not contribute any changes to
abstract state. In the cases of constant assignment, field
assignment, binary operator and object instantiation we update
the variable \( X \) with an empty set. In the case of lambda
assignment, we set \( X \) to be a singleton set. In case of variable
aliasing assignment, we set \( X \)’s mapping to the same as the
rhs. In case of lambda function invocation, we update the
mapping of the variable \( X \) with a union of all returnable
states from all the possible bindings of the variable \( G \) which

5 Obfuscation Potency Analysis

We analyze the potency of the CPS-based control flow ob-
fuscation.

an object of class \( C \), \( o \) be a value such that \( o.m(o) \) is well-
typed and terminating. Let \( M = \text{CMD}_{\text{cps}}[m] \). Let \( \text{geno} \in \text{GEN} \), \( \text{geno}' \in \text{GEN} \)
such that \( \text{geno} \prec \text{geno}' \). Then we have
\( M_{\text{SSA}}[m] o v \text{geno} \} = M_{\text{FJ}}[M] o v \text{geno}' \} \).
Control Flow Obfuscation for FJ using Continuation Passing

is bound to some lambda expressions. In case of method call, it is similar to the lambda function except that we look up the lambda expression from the global environment.

We overload the flow function for a lambda function declaration, whose output abstract state, will serve as the predecessor of the first statement in the function body.

$$\begin{align*}
\llbracket \lambda \rrbracket = \bigcup_{\text{caller}(\lambda)} \bot \llbracket a_1 \mapsto \text{eval}(\llbracket S \rrbracket, E_1^\lambda), \ldots, a_n \mapsto \text{eval}(\llbracket S \rrbracket, E_n^\lambda) \rrbracket
\end{align*}$$

where \{a_1, \ldots, a_n\} = \text{formalArgs}(\lambda). Given a statement \(S\) that calls \(\lambda, E^\lambda\), denotes the actual argument at \(i\)th position.

The helper function \(\text{caller}() : \Lambda \rightarrow \{\text{STATEMENT}\} \) returns the set statements in which the function \(\lambda\) is invoked. Let \(\delta\) denotes all the abstract states collected from all the statements of the target program.

$$\begin{align*}
\text{caller}(\lambda) &= \{S | S \in \text{STATEMENT} \land X \in \text{dom}(\delta) \land \\
&\lambda \in \sigma(S)(X) \land X(E) \in \text{rhs}(S) \text{ for some } E \}
\end{align*}$$

Helper function \(\text{eval}(\cdot, \cdot) : \text{STATE} \rightarrow \text{EXPRESSION} \rightarrow 2^\lambda\), takes an abstract state and returns a set of lambda functions which the expression might evaluate to.

$$\begin{align*}
\text{eval}(\sigma, c) &= \emptyset \\
\text{eval}(\sigma, \lambda) &= \{\lambda\} \\
\text{eval}(\sigma, X) &= \sigma(X) \\
\text{eval}(\sigma, E \text{ ap } E') &= \emptyset \\
\text{eval}(\sigma, E.F) &= \emptyset \\
\text{eval}(\sigma, \text{new } T()) &= \emptyset
\end{align*}$$

We apply the above analysis to our running example in Figures 2 and 3 until the abstract state reaches the fix point. We observe the following results.

| var  | func | var  | func | var  | func |
|------|------|------|------|------|------|
| raise | $\lambda_7$ | get_2 | $\lambda_78$ |  |
| get_3 | $\lambda_6$ | get_5 | $\lambda_32$ |  |  |
| pseq | $\lambda_8$ | pseq_raise | $\lambda_18$ |  |  |
| k_{18} | $\lambda_{70}$ | hd_{123} | $\lambda_{23}$ |  |  |
| k_{26} | $\lambda_{70}$ | raise_{28} | $\lambda_{79}$ |  |  |
| raise| $\lambda_7$ | k_{31} | $\lambda_{70}$ |  |  |
| k_{33} | $\lambda_{70}$ | raise_{35} | $\lambda_{23}$ |  |  |
| cond_{68} | $\lambda_8$ | visitor | $\lambda_{23}$ |  |  |
| raise_{62} | $\lambda_9$ | k_{52} | $\lambda_{30}$ |  |  |
| ploop | $\lambda_8$ | ploop_raise | $\lambda_{18}$ | first | $\lambda_{18}, \lambda_{78}$ |
| k_{45} | $\lambda_{15}$ | first_raise | $\lambda_{18}, \lambda_{78}$ |  |  |
| raise_{78} | $\lambda_{43}$ | k_{78} | $\lambda_{70}$ |  |  |
| tr_hd | $\lambda_{40}$ | hd_{ex_raise} | $\lambda_{33}$ |  |  |
| th | $\lambda_{30}$ | el | $\lambda_{52}$ |  |  |
| k_{90} | $\lambda_{70}$ | th_raise | $\lambda_{29}$ |  |  |
| id_bind | $\lambda_{17}$ | get_x | $\lambda_{32}$ |  |  |
| tr | $\lambda_{40}$ | hd_{17} | $\lambda_{23}$ |  |  |
| cond3 | $\lambda_9$ | cond5 | $\lambda_8$ |  |  |
| get_4 | $\lambda_6$ | get_6 | $\lambda_{28}$ |  |  |
| get_8 | $\lambda_{38}$ | get_9 | $\lambda_{38}$ |  |  |
| n_loop | $\lambda_{55}$ | n_second | $\lambda_{70}$ |  |  |
| seq | $\lambda_{67}$ | trycatch | $\lambda_{77}$ |  |  |
| ifelse | $\lambda_{68}$ | n_k_res | $\lambda_{15}$ |  |  |

For clarity and brevity, we adopt the following naming convention. We add line numbers to make common variables unique, e.g. \(\text{raise}_7\) denotes the raise from line 7.

As we can observe from the above, most of variables are given a unique lambda term to which they can be bound, except for \(\text{first}, \text{second}, \text{first_raise}\) and \(\text{second_raise}\). This is caused by the fact that the function \(\text{seq}\) is invoked in two different locations.

Through the analysis result, we can approximate a caller-callee relation between lambda abstractions. We reconstruct a global CFG of the obfuscated get by combining the call graphs and the local control flow graphs. The resulting CFG is presented in Figure 4.

As we discuss in the earlier section, the loss of precision is caused by the incompleteness of the context sensitive control flow analysis.

### 5.2 Context Sensitive Control Flow Analysis

A smarter attacker may attempt to uncover the CFG with better precision with context sensitive analysis.

In context sensitive analysis, we extend the abstract state with a context.

\[(\text{STATE}) \sigma \subseteq (\text{Variable} \times 2^\lambda) \cup \{\text{unreachable}\}\]

In this lattice, \(\text{unreachable}\) is the new \(\bot\).

We redefine the flow function \(\llbracket \cdot \rrbracket(\cdot, \cdot) : \text{STATE} \rightarrow \text{CONTEXT} \rightarrow \text{STATE} \rightarrow \text{STATE}\). The flow function takes a statement, a context and a state and returns an updated state. Given a statement \(S\) and a context \(c\), we write \(\llbracket S \rrbracket(c)\) to denote \(\llbracket S \rrbracket(c)(\sigma_S)\) by making \(\sigma_S\) an implicit argument where

\(\sigma_S = \text{join}(c, S)\)

Recall from on our running example, the imprecision of the context insensitive analysis is caused by the two calls of \(\text{seq}\) in lines 12 and 13 in Figure 2. If we define the context to be last call sites of the function, i.e. program locations, we would achieve a better precision.

### 5.3 Complexity of Sub-graph isomorphism

Regardless of the precision of the static analysis result, it is computationally expensive to match the original control

As we discuss in the earlier section, the loss of precision is caused by the incompleteness of the context sensitive control flow analysis.
flow graph with the approximated control flow graph in general. Let the original CFG to be $H$ and the approximated CFG to be $G$, we want to check whether $H$ is sub graph isomorphic to $G$, which is NP-complete [6]. For instance Ullmann’s algorithm [18] is known to be exponential. Some improvement with heuristic algorithms exist. There is no known algorithms solving this problem in polynomial time. This check only returns yes or no. Finding all possible isomorphic sub-graphs leads to sub graph matching problem, which is also NP-Complete.

Note that some linear algorithm exists for the special case in which one of the input graphs is fixed and the other is a planar graph. Unfortunately CFG generated from Java in general is not guaranteed to be a planar graph [17].

6 Related Works
The CPS-based control flow obfuscation is rooted from the connection between SSA forms in imperative programming languages and lambda terms in functional programming languages [2, 4, 9]. Our translation scheme is an extension of Lu’s work[11] and is inspired by Kelsey’s work [9]. In contrast with Lu’s work, we are targeting FJ instead of C style language. As an improvement to Lu’s work, our translation scheme supports exception handling, recursive call and call to methods within the same scope with continuations. In contrast to Kelsey’s work, our translation is targeted at an imperative language extended with higher order function instead of Scheme. Giacobazzi et al proposed a method to construct general obfuscators using partial evaluation with distorted interpreters [7]. Their work provides a uniform reasoning of how attacks using abstract interpretation can be foiled by a particular obfuscation method (by constructing a specific distorted interpreter). Ancona and Corradi [1] formalized SSA form for FJ. They applied SSAFJ to improve the type analysis of object oriented languages such as Java.

7 Conclusion
We extend and develop CPS-based control flow obfuscation for FJ with exception handling. We formalize the strategy as a source to source translation scheme. We show that the control flow obfuscation technique is effective against attacks using static control flow analysis, in particular context insensitive analysis. We are in the process of implementing the reported technique. The progress and some examples can be found in our development repository [12].

References
[1] Davide Ancona and Andrea Corradi. 2016. A formal account of SSA in Java-like languages. In Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs, FTfJP@ECOOP 2016, Rome, Italy, July 17-22, 2016, Vladimir Klebanov (Ed.). ACM, 2. 
https://doi.org/10.1145/2955811.2955813
[2] Andrew W. Appel. 1998. SSA is Functional Programming. SIGPLAN Not. 33, 4 (April 1998), 17–20. 
https://doi.org/10.1145/278283.278285
[3] Ian Capparell and Bart Preneel. 2010. A General Model for Hiding Control Flow. In Proceedings of the Tenth Annual ACM Workshop on Digital Rights Management (DRM ’10). ACM, New York, NY, USA, 35–42. 
https://doi.org/10.1145/1866870.1866877
[4] Manuel M. T. Chakravarty, Gabriele Keller, and Patryk Zadarnowski. 2003. A Functional Perspective on SSA Optimisation Algorithms. In COCV ’03: Compiler Optimization Meets Compiler Verification (Electronic Notes in Theoretical Computer Science). Jens Knoop and Wolf Zimmermann (Eds.), Vol. 82. Elsevier Science, 347–361. Issue 2. 
https://doi.org/10.1016/S1571-0661(05)82596-4
[5] Jien-Tsai Chan and Wuu Yang. 2004. Advanced obfuscation techniques for Java bytecode. Journal of Systems and Software 71, 1 (2004), 1–10. 
https://doi.org/10.1016/S0164-1212(02)00066-3
[6] Stephen A. Cook. 1971. The complexity of theorem-proving procedures. In STOC. ACM, 151–158.
[7] Roberto Giacobazzi, Neil D. Jones, and Isabella Mastroeni. 2012. Obfuscation by Partial Evaluation of Distorted Interpreters. In Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation (PEPM ’12). ACM, New York, NY, USA, 63–72. 
https://doi.org/10.1145/2103746.2103761
[8] Guardsquare. 2020. ProGuard: Open Source Optimizer for Java and Kotlin. 
https://www.guardsquare.com/en/products/proguard
[9] Richard A. Kelsey. 1995. A Correspondence Between Continuation Passing Style and Static Single Assignment Form. In Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations (IR ’95). ACM, New York, NY, USA, 13–22. 
https://doi.org/10.1145/202529.202532
[10] Timesz László and Ákos Kiss. 2007. Obfuscating C++ Programs via Control Flow Flattening. In 10th Symposium on Programming Languages and Software Tools (SPLST 2007). 15–20.
[11] Kenny Zhuo Ming Lu. 2019. Control flow obfuscation via CPS transformation. In Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2019, Calcium, Portugal, January 14–15, 2019, Manuel V. Hermenegildo and Atsushi Igarashi (Eds.). ACM, 54–60. 
https://doi.org/10.1145/3294032.3294083
[12] Kenny Zhuo Ming Lu. 2020. Control flow obfuscation for Java code. 
http://github.com/luzhuomi/obsidian/
[13] Anders Møller and Michael I. Schwartzbach. 2018. Static Program Analysis. Department of Computer Science, Aarhus University, 
http://cs.au.dk/~amoeller/spa/
[14] Oracle. 2020. Oracle Java Technologies. 
https://www.oracle.com/java/technologies/
[15] K. Petersson. 1990. Syntax and Semantics of Programming Languages.
[16] Davide Pizzolotto and Mariano Cecatto. 2019. Obfuscating Java Programs by Translating Selected Portions of Bytecode to Native Libraries. CoRR abs/1901.06942 (2019). 
http://doi.org/10.1109/SCAM.2018.00012 arXiv:1901.09492
[17] Raphael. 2011. On Planarity of Control Flow Graphs. 
http://imazy.verrech.net/2011/10/on-planarity-of-control-flow-graphs/ 
[18] J. R. Ullmann. 1976. An algorithm for subgraph isomorphism. JOURNAL OF THE ACM 28, 1 (1976), 31–42.
[19] Balachandran Vivek, Sufatrio, Tan Darell, and Thing Vrizlynn. 2016. Control flow obfuscation for Android applications. Computers & Security 61 (05 2016). 
https://doi.org/10.1016/j.cose.2016.05.003
[20] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report. Charlottesville, VA, USA.

A Appendix
A.1 Pre-processing step that fix while block that has multiple entry labels
The only possible case that violates the restrictive form is the use of try-catch with a while loop in the handler.

try {
    
    Li : throw new Exception();
}
The above can be converted into the restrictive form by inserting an empty assignment block in front of the while block.

```java
try {
    ...
    Li : throw new Exception();
    ...
    Lj : throw new Exception();
} catch (Exception e) {

    Lk: {
    
    Ll: join (x = phi(Lk:xk, Lm:xm)) while (e) {
        Lm: ...
    }
}
```