# A General Model for Hiding Control Flow

Jan Cappaert
Katholieke Universiteit Leuven
Dept. Elect. Eng.-ESAT/SCD-COSIC
Kasteelpark Arenberg 10, 3001 Heverlee, Belgium
jan.cappaert@esat.kuleuven.be

Bart Preneel
Katholieke Universiteit Leuven
Dept. Elect. Eng.-ESAT/SCD-COSIC
Kasteelpark Arenberg 10, 3001 Heverlee, Belgium
bart.preneel@esat.kuleuven.be

## ABSTRACT

This paper proposes a general model for hiding control flow graph flattening in C programs. We explain what control flow graph flattening is and illustrate why it is successful as protection against static control flow analysis. Furthermore, we propose a scheme, complementary to control flow graph flattening, which does not leak any control flow graph information statically. Instead of relying on ad hoc security by using variable aliasing and global pointers to complicate data flow analysis of the switch variable, we try to base our security claims more on information theory, data flow, and cryptography. Our formal model is structured and extendable. Moreover, it can specify which minimum of information to hide from the program (e.g. a secret value or function) such that no control flow information is leaked. To express the robustness of our scheme we present some attacks and their feasibility. Finally, we sketch a few scenarios in which our solution could be deployed.

## Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; D.2.11 [**Software Engineering**]: Information hiding

## General Terms

Design, security, theory

## Keywords

Software protection, static analysis, code obfuscation, cryptography

## 1. INTRODUCTION

The last two decades software protection has been a highly investigated research area. Protection against analysis and tampering are hot topics in both industry as the academic world. On the the other hand, control flow analysis is also of importance in many applications. It is used for numerical optimizations, both in speed (branch prediction, peephole optimizations, hot code analysis) as in size (e.g. dead code analysis). Furthermore, many other analyses (e.g. data flow analysis) are flow-sensitive and rely on accurate control flow information. Malicious users also perform control flow analysis on programs. They might want to steal an algorithm or data embedded in a program. Sometimes, adversaries tamper with the functionality of a program to achieve more privileges. Hence, in certain scenarios we would like to protect our program from control flow analysis.

In this paper we focus on protection against static analysis. We will illustrate that protecting against static analysis also has its applications, and that our model is highly suitable in the context of hardware support (smart card, dongle, TPMs, ...). In contrast to several other control flow graph flattening techniques we will illustrate that our method is able to hide the entire control flow (and other program internals) behind a single value or function. We also focus on protection against "local attacks", as the attacker usually 'jumps' directly to a particular piece of interest. On a source level for example C is mostly not flow-sensitive, while on a binary level code is more abstract. An incorrect jump in a binary can trigger a (slight) desynchronization of the disassembler [1].

Our paper is structured as follows: Section 2 explains control flow graph concepts and why flattening helps to protect against analysis. The next two sections present our main contributions to this research area. Sections 3 presents a scheme that forces an adversary to perform global analysis, even if he interested in a particular control transfer, while Section 4 presents application scenarios where the control flow can be hidden behind a key or in hardware. To validate our results, we present some attacks in Section 5 and Section 6 compares our results to other research. Finally we summarize our work and draw conclusions in Section 7.

## 2. CONTROL FLOW GRAPH FLATTENING

### 2.1 The Control Flow Graph

Consider a program $P$ as a graph $G = \langle V, E \rangle$, where $V$ represents the set of basic blocks $\mathrm{BB}_1, \mathrm{BB}_2, \ldots, \mathrm{BB}_n$ and $E$ represents the set of control flow edges $e_1, e_2, \ldots, e_m$. This is usually called the control flow graph (CFG) of $P$. Conventional programs usually contain hard-coded information denoting how and where control can be transferred from one to another block. If we collect all this information (statically) and represent it graphically we get a static control flow graph, where nodes represent basic blocks and edges represent control transfers. If we trace a program dynamically, we can reconstruct a part of this graph, built out of a subset of $V$ and $E$.

## 2.2 CFG Flattening

In a flattened version of a program, a dispatcher node DN gives control to a specific basic block $\mathrm{BB}_i$ [3]. After executing a block, control is transferred back to the dispatcher node DN, who decides how to continue. At every iteration, any block is a possible candidate to be selected next. Because the switch makes decisions based on a switch variable, the control flow problem has become a data flow problem. At first glance, it is usually not clear in which order the basic blocks will be executed (unless the dispatcher node is easy to analyze and/or the control flow information is still hardcode in the basic blocks). Whereas in a traditional program most basic blocks during execution can give control to 1 or 2 other basic blocks (2 in the case of a decision point), control in a 'flattened' program can be transferred to $n$ other points, $n$ being the number of basic blocks. Not performing data flow analysis (e.g. constant propagation) would yield in $O(n^k)$ possible execution paths, where $k$ denotes the number of executed blocks.

Figure 1(a) shows a flattened CFG implemented as a switch structure, nested in a loop. DN represents the dispatcher node, $\mathrm{BB}_i$ the basic blocks, and $l_i$ the labels. The switch structure decides which basic block to execute based on which label value is assigned to the switch variable. If a label $l_i$ is not present, control will be transferred back to the dispatcher node DN. This is done via a sort of fall-through path. Consider $L = \{l_1, l_2, \ldots, l_m\} \subset \{0,1\}^n$ the set $n$-bit labels in a program, then $L_*$ matches all remaining labels, namely $\{0,1\}^n \backslash L$. The basic block $\mathrm{BB}_*$ is a special basic block that will be executed each time the switch falls through because no match to the value of the switch variable is found in $L$. In C for example, it is possible to specify a basic block in this fall-through path, using the `default` label. Figure 1(b) shows a nested `if-else` CFG which offers equivalent functionality as the flattened CFG. [1]

## 3. STRENGTHENING CFG FLATTENING

In this section we gradually present our protection scheme. Step by step we perform improvements on a flattened control flow graph to make analysis harder.

### 3.1 Setup Phase

To illustrate our method of transforming a program in a flattened one, which is hard to analyze statically, we start with the implementation as published by László and Kiss [7]. In this model a single variable, the "switch variable" (`swVar` in the listing below), contains a label value used to switch control to the corresponding basic block. In each basic block new values are assigned to this variable. The following example was used from their paper:

```
case 2:
  if (i <= 100)
    swVar = 3;
  else
    swVar = 0;
```

[1]We assume that a basic blocks does not "fall through" to the next basic block, which would be the case if one omits the `break` statement in C's `switch` structure.

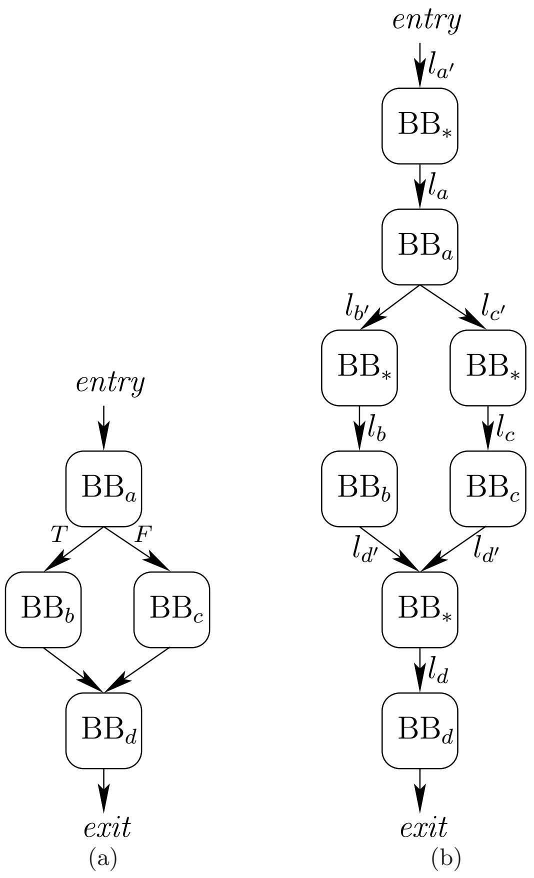


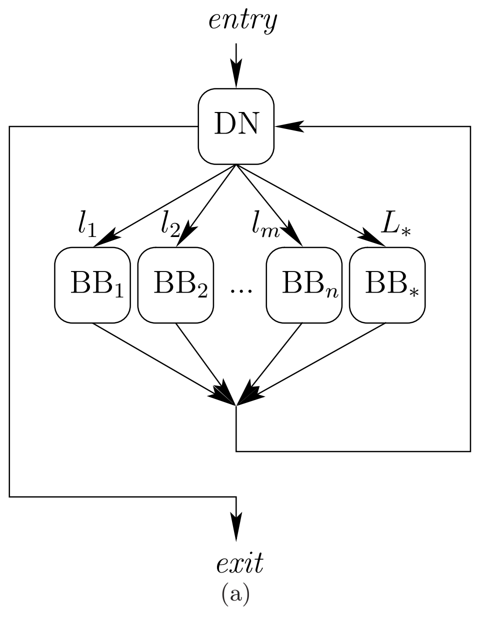
**Figure 2: (a) A simple `if-else` branch CFG. (b) The transformed "graph" according to our one-way function solution.**

First, note that constants are assigned, which should be hard-coded labels if no fall-through path is present (i.e. the `default` label in C). Secondly, the branch still contains an `if-else` construct. Hence, the code still reveals where execution will branch. The assignments in this example are of the form $x = l_i$ where $x$ represents the switch variable and $l_i$ denotes the label value of the next basic block to execute. This allows an attacker to perform "local attacks". To deduce the next block, one just has to look up the block appointed by label $l_i$. While to find all possible preceding blocks, an attacker has to go over all basic blocks and scan for assignments of the form $x = l_i$.

In the following sections we propose steps to increase security of this control flow graph flattening algorithm. Our goal is to force the attacker to do a global analysis, even if he wants to perform a local attack.

### 3.2 Step 1: Remove Hard-Coded Control Flow Values

As the implementation as proposed by László and Kiss is susceptible to local analysis attacks, we promote the use of relative updates. Each assignment to the switch variable is a relative update of its current value. The example above could look like this:



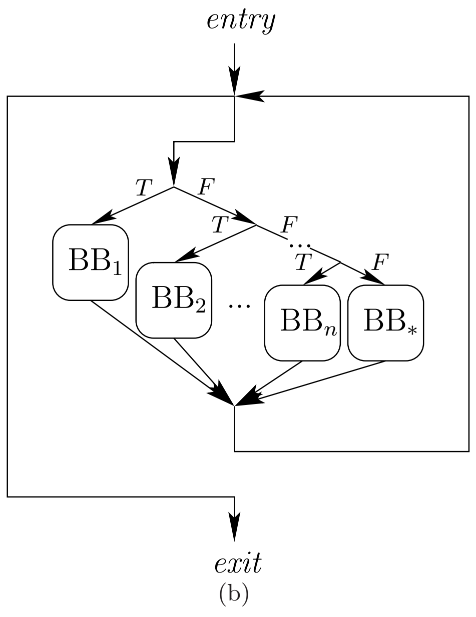


**Figure 1: (a) A flattened control flow graph. (b) A `if-else if-else if-...` structure, which is equivalent to the flattened program in (a).**

```
case 2:
  if (i <= 100)
    swVar = swVar + 1;
  else
    swVar = swVar - 2;
```

As such control flow information is less explicit. Assignments of the form $x = x + \alpha$ yield that possible targets (successor blocks) can only be derived if the actual value of $x$ is locally available. If not the case, the attacker should perform backward analysis to compute predecessor blocks, and for each of these again predecessor blocks, etc. This yields that a simple local attack requires a global analysis with worst case complexity $O(n^k)$ with $k$ the number of preceding blocks, leading to the particular block, and $n$ the number of basic blocks.

## 3.3 Step 2: Use a Single Uniform Statement

As mentioned earlier, several basic blocks in LŚaló and Kiss's implementation still contain branches: the `if-else` reveals a decision point. To replace these hard-coded branches, we propose the use of a single uniform statement. A branch namely depends on a condition which always returns *true* or *false*. With the appropriate logic we can map *true*/*false* values on 1/0. These values can then be used in an expression. The example above could be reduced to:

```
swVar = swVar + 1 - (i <= 100) * 3;
```

We get one uniform statement, which can be represented as a function. This function we will later call the *branch function* $B()$:

$$B(x) = x + \alpha + y \times \beta$$

Here $x$ denotes the switch variable, and $y$ denotes the condition. Both are represented by $n$-bit variables, but due to the outcome of the condition $y$ is always of the form $0000000a$ where $a$ is the Boolean result 0/1 of the conditional expression. '+' and '×' just represent addition and multiplication in GF($2^n$). Furthermore, we notice two hard-coded constants $\alpha, \beta \in \{0,1\}^n$. $\alpha$ and $\beta$ can always be chosen such that for any basic block $BB_s$ and any two target blocks $BB_{t_1}$ and $BB_{t_2}$:

$$\forall l_s, l_{t_1}, l_{t_2}, \exists \alpha, \beta : l_{t_1} = l_s + \alpha \text{ and } l_{t_2} = l_s + \alpha + \beta$$

Jump statements, such as C's `goto`, `break`, or `continue`, only transfer control to 1 successor block. Hence, they have a simpler form: $B(x) = x + \alpha$. To increase confusion, one could add opaque predicates [2], i.e. conditional expressions which are known at compile-time but which are hard to compute statically. Once the predicate is added, we get a similar form as above. In this paper we only consider always-*true* and always-*false* predicates whom outcome we will map onto $\{0, 1\}$.

## 3.4 Step 3: Use Non-Local Values

While the previous step seems to be an improvement in terms of def-use chains (the switch variable is always defined in the previously executed block, which is not known at first glance in a flattened CFG), local analysis is still trivial. Namely, the value contained by the switch variable is always one of the labels leading to the basic block containing this code. Hence, it is locally available. To solve this, we propose to use an earlier computed value. For example, consider the label of the previous block. During a static attack the adversary does not know which block preceded the other block, unless he solves the statements in each basic block, which themselves depend on other values from other blocks, etc. Instead of $x_{i+1} = x_i + \alpha + y \times \beta$ where $x_i$ is locally available, we would get $x_{i+1} = x_{i-1} + \alpha + y \times \beta$

However, if $x_{i-1}$ in block $BB_a$ contains the label of the previously executed block, we might have multiple possible label values if several blocks can give control to $BB_a$. If $BB_a$ in turn gives control to another block $BB_b$, we cannot just rely on a single value contained by $x_{i-1}$. To solve this we need to:

- create two labels for $BB_b$ (two paths towards it), however this only propagates the problem, or
- introduce a corrector value, or
- use a dummy basic block to jump to (all predecessor blocks) and use the label of that block.

We opted for the last solution. The dummy block then transfers control to $BB_b$ without passing information from its predecessor blocks. While this seems to suggest a $x = l_i$ construction, it is not. We propose to use a label value which is not hard-coded hence not susceptible to static analysis. For this, the fall-through path of a switch becomes very handy. In this basic block we could specify how to update the switch variable and store its current (runtime) value for use in the next basic block. As a consequence, two or more blocks can (eventually) give control to a common successor block, by first giving control to the same (or even different) dummy basic block(s). These dummy blocks will just "fall through" to the targeted block.

From now on we will denote our one-way function as the *transition function* $F()$ as each control flow transfer a label value flows through this function. For the security of our overall scheme we propose the following properties for $F()$:

**One-way** This property computationally withholds the attacker from doing backward analysis. For example, to get the previous value of the switch variable in a *use* site, one has to do backward liveness analysis to located the *def* site.

**Bijective** In a bijection, each unique label will be mapped onto another unique label. The importance of this property will become clear in Section 5.1.

**Diffusion** Preferably $F()$ has good diffusion properties such that constant or range propagation [13] becomes significantly harder.

Examples of one-way functions, and thus good candidates for our transition function, include:

**The discrete logarithm** The discrete logarithm over a finite field GF($p$). In GF($p$) we can use $F(x) = g^x \bmod p$ with $g$ generator of the field and $p$ a large prime. It is hard to compute $x$ for a given $F(x)$. Furthermore, this function is bijective (for GF($p$)$\setminus\{0\}$). If we also specify our branching function in GF($p$) we get: $B(x) = (x + \alpha + y \times \beta) \bmod p$ with constants $\alpha, \beta \in \text{GF}(p)$ and $y \in \{0,1\} \subset \text{GF}(p)$. In this case $B()$ is also a bijection, and invertible. The latter property of $B()$ is required to compute $\alpha$ and $\beta$.

**Cryptographic hash functions** Hash functions in cryptography are typically used as compression functions from $m$ to $n$ bits. However, we can also use them for $n$-to-$n$ bit conversions. Cryptographic hash functions are crafted in such way that they are fast, but still hard to invert. Strictly speaking cryptographic hash functions have collisions, and thus they are not bijective. However, finding collisions is considered to be hard, such that for the purpose within our protection scheme we can consider them as a good candidate. The proof in Section 5.1 elaborates on this. Defining our transition function would yield: $F : \{0,1\}^n \rightarrow \{0,1\}^n : F(x) = hash(x)$, while a bijective branching function could be: $B(x) = x \oplus \alpha \oplus y \times \beta$.

For the rest of the paper, we will implicitly consider the second example, where $F()$ is a cryptographic hash function, and where $B(x)$ uses exclusive OR as binary operator. Section 5.3 reasons on why it is so important to choose a binary operator with good properties. We will also use the term 'preimage', which is the inverse image of a hash value $y = F(x)$: $F^{-1}(y) = x$. Finding preimages for hash functions is a hard problem and one of the basic properties of this cryptographic primitive.

## 3.5 Overall Structure

To summarize, our scheme looks as follows:

- the fall-through basic block $BB_*$ contains:
  - a new variable storing the runtime value of the switch variable (which lead to $BB_*$ and which is not hard-code in the code): $z = x$, and
  - a call to our transition function: $x = F(z)$.
- all other basic blocks $BB_i$ contain:
  - an instantiation of $B()$ with precomputed $\alpha$ and $\beta$. This $B_i()$ operates on the "runtime label" stored in $z$, which is from a static point of view a secret value: $x = z + \alpha_i + y_i \times \beta_i$. Remember that the label value in $z$ is the preimage of the label that is used to transfer control to $BB_i$.

The transition function $F()$ is used for its one-way and diffusion properties. At runtime, it will allow a dynamic label to "fall through" to a static label. The branch function $B()$ is used to transfer control to other blocks. This is done via a dummy block, linked to by the dynamic label.

Finally, we get a structure that looks like the example in Figure 2(b). Figure 2(a) illustrates the CFG of a simple `if-else` branch. Here control can be branched from $BB_a$ to $BB_b$ or $BB_c$. The transformed control flow graph (after removing the flattening, and ommiting passes through the distribution node DN) is illustrated in Figure 2(b). Labels with a dashed index are preimages (and thus runtime values) of hard-coded labels. The $BB_*$ blocks all represent the same block, namely the block that matches all label values that are not hard-coded in the switch. Figure 3.6 illustrates how these labels –the control flow data– flows through the iteration $F()$ and branch functions $B_i()$. In this figure, the labels in a circle are hard-coded in the program, while the other labels represent runtime values.

When applying such a protection scheme on a program's control flow, the defender can choose all labels at will and compute the required hard-coded constants, such that relations in the flattened graph still represent relations of the original control flow graph. Because $F^{-1}()$ is hard to compute, the defender has to generate a set of small hash chains in advance. From this set he then picks his labels and their preimages. If values are chosen randomly, attackers cannot exploit any patterns.

So far our scheme forces an adversary to perform global analysis if he wants to investigate local control flow. In the following section we sketch scenarios in which we can hide a program's control flow (and other internals relying on it) entirely, such that even global analysis techniques cannot be applied.

## 3.6 Step 4: Make Values Hard to Compute

While it is already hard in a flattened control flow graph to analyze what a predecessor block was, we can also make it computationally hard for the attacker to retrieve $x_{i-1}$ out of $x_i$. For this, we could for example use a one-way function which makes it computationally hard for the attacker to compute the preimage of a label statically. Consequently, the defender has to precompute these hash pairs himself.

## 4. APPLICATIONS

In this section we describe application scenarios of our protection scheme. To hide control flow statically we just apply our scheme of Section 3 and either split off a portion of the program (data or code), or we rely on a (statically) hard problem. First, a program can be split into a program and a key. Without the secret key, the program will not run correctly and more important: nothing can be learned by static analysis. In a second scenario, the program splits off a small function which can be protected in hardware, such as a dongle or a smart card. And last, we present a "stand alone" solution that computes a key at runtime, but still makes it hard to derive the key statically (to eventually perform control flow analysis).

We strongly believe that protection against static analysis still has its applications. First of all, static analysis will provide a complete overview of the program, while dynamic analysis only provides the adversary a trace of the executed path. To extend this partial information, an attacker might combine dynamic analysis with some static analysis techniques. However, due to its construction, our scheme restricts the capabilities of both forward and backward analysis. Secondly, attackers might not know the inputs to the whole program or to parts of it. One way to proceed would be to guess these values, however this does not guarantee that the path of interest gets executed. In the first two application scenarios we propose the attacker cannot run the code without an additional secret component. Random guessing of the switch variable values will not lead to extra knowledge of the targeted code, as will be illustrated in Section 5.1. Only in the third scenario, all code is in place to run the flattened program part. However, due to the nature of opaque predicates, the attacker is challenged to prove that the generated label value (and thus the execution as well) will always be the same.

## 4.1 A Single Secret Key

A first example, and maybe the most logical is to hide a program's secrets behind a secret key. This idea is used for centuries as the main primitive in cryptography, where message confidentiality is protected by a (usually smaller) secret key. Once we succeed in hiding a program's control flow we can make other analyses flow-sensitive to reduce their efficiency.

The big advantage of this key scenario is that we can send

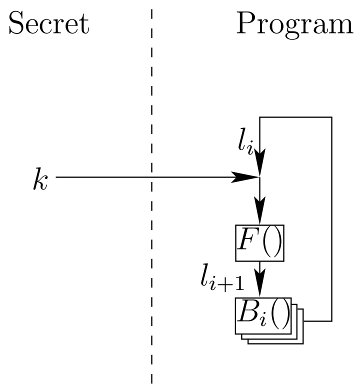


**Figure 4: A scenario with just a single secret value, the *key*.**

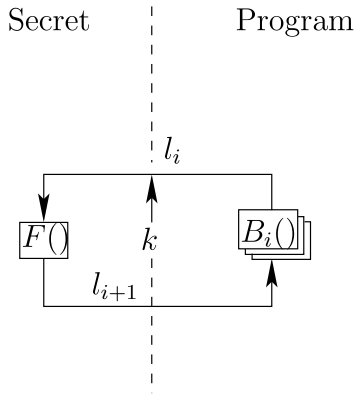


**Figure 5: A scenario with a single *function* implemented in hardware or ran remotely on a server.**

applications in advance of sending the key. However, if we rely on secrecy of a key, we can only run the program once. As from then on, the functionality is revealed. Our technique however has some advantages over pure bulk encryption. It allows to merge programs (each key $k_i$ will yield execution of a program $P_i$). These programs can even have basic blocks in common such that their actual CFGs will we interweaved.

In our protection scheme we force the attacker to do global analysis, starting at the entry point of the CFG. Consider the label of the first basic block $l_1$, then the secret key $k$ can be a preimage value of $l_1$: $k = F^{-1}(l_1)$.

A practical scenario could be remote voting applications on PCs as this concerns programs that execute once (after installation). The scenario could look as follows: one big program is installed in advance, at moment $t_1$ a key $k_1$ is sent to execute functionality $f_1()$, at moment $t_2$ another key $k_2$ for $f_2()$, etc. Keys could be sent via the Internet from one or more servers.

## 4.2 A Secret Function

Our scheme uses two types of functions: the branching functions $B_i()$ and the transition function $F()$. Basically, the branching functions contain all the control flow information, needed for branching and jumping. The transition function,however, is needed to make our scheme stronger (making data flow analysis one-way).

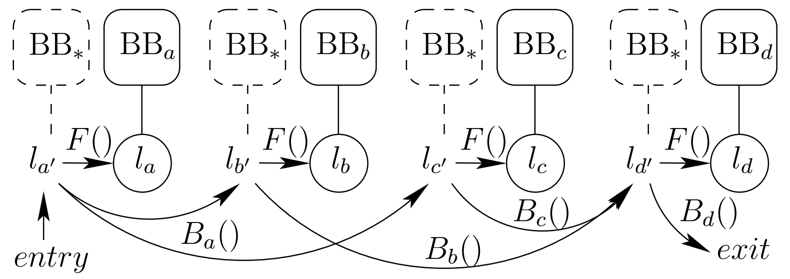


**Figure 3: How control flow data flows through transition function $F()$ and four branch functions: $B_a(), B_b(), B_c()$, and $B_d()$.**

In this scenario we rely on a trusted 'party' that has computational abilities. As such it can compute $F()$ in a challenge-response alike way. This 'party' could be a server, but also a smart card or a hardware dongle. Without this additional piece, the software cannot run and attackers cannot learn from it.

### 4.3 Stand Alone Solution

Apart from the applicability of the models above, we also propose a "stand alone" application where control flow is protected by a secret value which is computed at runtime. Assuming that stealthy strong opaque predicates [2] exist, we can construct a 'random' secret $n$-bit value $k$ by using one of the following constructions:

$$k = p_1 \parallel p_2 \parallel \cdots \parallel p_n$$

where $p_i$ represent opaque predicates and $\parallel$ denotes the concatenation of bits. The disadvantage of this combination is the bitwise combination. If one or more predicates are broken, bits are revealed. In our scheme however, this value is first processed by $F()$ which has good diffusion properties, and thus reduces the risk of being exploitable. An alternative construction is:

$$\begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_n \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & \dots & r_{1n} \\ r_{21} & r_{22} & \dots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ r_{n1} & r_{n2} & \dots & r_{nn} \end{pmatrix}$$

or $K = PR$ with $R$ a hard-coded $n \times n$ random matrix of full rank over GF(2) and $K$ and $P$ matrices with dimension $n \times 1$ where $P$ is the vector composed of all $p_i$ bits, which come from independent opaque predicates. To a static attacker this should have the same probability as a random bit. Dodis *et al.* use a similar technique to 'blend' two sources of randomness [11].

It should be clear that this scenario makes static analysis hard, but does not prevent an attacker from just running the code. He even might repeatedly run the code with different inputs and bypass the predicates as he always observes generation of the same key.

## 5. SCHEME ATTACKS

### 5.1 Brute Force Attacks

Even though in static analysis the inputs are not known, several global analyses succeed in extracting information. Techniques include: constant propagation, range propagation [13], ... Also, techniques such as abstract interpretation have been proven useful, e.g. for breaking opaque predicates [5].

We will prove that many of the strength of these techniques does not apply to our models. First, consider the model where a function is secret. As $F() : \{0,1\}^n \to \{0,1\}^n$ is unknown, the attacker cannot propagate constants (or ranges). Secondly, consider our model where we either keep a key $k$ secret, or construct it at runtime using opaque predicates. If the $n$-bit key $k$ is uniformly distributed, its entropy $H(k)$ equals $n$ bits. We will show that brute-forcing any of the intermediate values of the switch variable, has the same chance of succeeding as long as the switch variable's entropy is preserved during execution.

PROOF. Consider the model in GF($2^n$), where $F(x) = hash(x)$ is a strong hash function, and $B(x) = x \oplus \alpha \oplus y \times \beta$.

Remember that $y \in \{0,1\}$ as $y$ represents the mapped outcome of a condition (or predicate). With probability $P(y = 0)$ we get $B(x) = x \oplus \alpha$, while with probability $P(y = 1)$ we get $B(x) = x \oplus \alpha \oplus \beta$ with $\alpha, \beta$ constants. As exclusive OR has a balanced outcome, it is not loosing information. Consequentially, $x \to B(x)$ is a bijection. It maps GF($2^n$) to GF($2^n$). Furthermore, bijections preserve entropy because probabilities are just mapped to other values. Hence, we conclude: $B(x)$ preserves entropy of the switch variable during program execution.

For the hash function, it is somewhat different. Strictly speaking cryptographic hash functions are not fully bijective. However, finding collisions for good hash functions is a hard problem (computationally expensive) . As such, we can conclude that our attacker cannot exploit these collisions. We consider therefore $hash(x)$ also bijective for the inputs we use. And thus it also preserves entropy. □

Note: for GF($p$) we can provide a similar proof. It is more sound though, as $B(x) = (x + \alpha + y \times \beta) \bmod p$ is a full bijection, as well as $F(x) = 2^x \bmod p$ (for GF($p$)$\setminus\{0\}$).

### 5.2 Attacking the Transition Function

When the transition function $F()$ is known, one can generate the hash chains offline. For $n$ bits this implies $2^n$ values. Using these chains one can search for preimages of a certain (hard-coded) label $l_i$, allowing to solve the branch function equation based on (the preimage of) this label.

However, this full computation of $F()$ becomes impractical for large numbers of $n$. Consider $2^{10}$ hard-coded 32-bit labels. This yields the probability of finding a preimage of one label $2^{10}/2^{32} = 1/2^{22}$. First, due to the one-way properties of $F()$ the attacker would only be able to propagate this value forward, only reconstructing a part of the static control flow graph. Secondly, for cryptographic hash functions an output of 160 bits is recommended. Thus, even if there were millions of basic blocks, an attack would still require $2^{140}$ attempts to find a usable preimage value.

## 5.3 Attacking the Branch Function

If $B()$ is a weak construction, our scheme becomes vulnerable to attacks as well.

First, consider the use of bitwise ANDs or ORs in $B()$ instead of the exclusive OR. Then $B()$ would lose information as AND and OR are lossy. On average $x \rightarrow x \,\&\, \alpha$ (with $\alpha$ constant, and '&' bitwise AND), loses 50% of information (i.e. half of the bits are cleared due to zeros in $\alpha$). Applying this repeatedly would yield very low entropy allowing brute force attacks.

Secondly, while bitwise binary operators are fast, they propagate differences. If an attacker detects biases he can propagate through $B()$. Nevertheless, we still have $F()$, which in case of a one-way function should have good diffusion properties. One could also use other binary operators that are invertible and bijective. For example, operators based on T-functions [12].

A third attack, is an attack on a reduced version of our scheme. Namely, one where the fall-through path (`default:`) is never used. Consider the implementation as is in Step 2 (see Section 3), neglecting the fact that label values of an executing block are visible in the case label. Consider the equation:

```
swVar = swVar + 1 + ((cond) * 3);
```

Here we can deduce that the difference between target label $l_1$ and $l_2$ is 3. Consider a trivial flattened program with labels $L = \{1, 2, 7, 10\}$. The only possible target labels where label '7' and label '10'. In a real program, with $m$ labels we would have $O(m^2)$ possible differences. It is obvious that the more basic blocks (and thus labels) a flattened program has, the more complicated this difference matching becomes.

From Step 4 on, our scheme is no longer vulnerable to this difference attack on labels due to the fact that we no longer jump to hard-coded labels, but to their preimages. These values are not hard-code, but they are matched by the fall-through label $L_*$. However, due to reduncancy when control flow converges, one can craft a similar attack. Consider the branch in Figure 2(b). As both $\text{BB}_b$ and $\text{BB}_c$ give control to $\text{BB}_d$, one can deduce that their branch functions map to the same preimage. Consequently, $l_{b'} \oplus \alpha_b = l_{c'} \oplus \alpha_c$. Furthermore, does the branch function in $\text{BB}_a$ reveal that $\beta_a = l_{b'} \oplus l_{c'}$. Solving this set of equations gives: $\beta_a = \alpha_a \oplus \alpha_b$. The complexity is equivalent to the one of the example above: $O(m^2)$ for $m$ hard-coded $\alpha$ values. Several solutions exist:

**Split basic blocks** Splitting basic blocks increases the amount of $\alpha$ values leading to a $\beta$ value. Hence, the compexity will also grow from kwadratic to exponential in the amount of basic blocks in both paths.

**Opaque predicates** As mentioned earlier, opaque predicates [2] allow us to transform each branch function $B_i()$ to the form $B(x) = x \oplus \text{cst}_1 \oplus (p) \times \text{cst}_2$ with $p$ an opaque expression. If the attacker cannot distinguish $p$ from a real expression, he will assume $\text{cst}_2$ to be $\beta$, and $\text{cst}_1$ $\alpha$ while this is not the case. In case of an always-true predicate $\alpha$ can be split into two components, while in case of an always-false predicate $\text{cst}_2$ will present a fake $\beta$. This also increases the matching complexity.

We propose a stronger solution. Let the branch function in $\text{BB}_b$ map to $l_{d'}$, while the one in $\text{BB}_c$ maps to $l_{d''} = F^{-1}(l_{d'})$. Hence, a target label value is never used twice. Instead, in case of converging control flow, one jumps to the preimage of that target label value.

## 5.4 Other Attacks

While all previous attacks focus on exploiting our protection scheme, we want to indicate for completeness that adversaries also can use other information to guess the execution order of basic blocks. Examples include: functions (`fopen` comes usually before `fclose`), variables names and liveness, etc. Hence it always better to apply this technique in combination with other techniques such as variable renaming, dereferencing of variables, function pointers, and other obfuscation techniques.

## 6. RELATED RESEARCH

## 6.1 Tamper Resistant Software

In '96 Aucsmith [6] illustrated in his paper a scheme to implement tamper-resistant software. His technique protects against analysis and tampering, both static and dynamic. For this, he uses small, armored code segments, also called integrity verification kernels (IVKs), to verify code integrity. These IVKs are protected through encryption and digital signatures such that it is hard to modify them. Furthermore, these IVKs can communicate with each other and across applications through an integrity verification protocol. Many papers in the field of tamper resistance base their techniques on one or more of Aucsmith's ideas.

Several years later, Ge *et al.* published a paper on "control flow based obfuscation" [8]. Although the authors titled their work as a contribution to obfuscation, the control flow information is protected with an Aucsmith-like tamper resistance scheme.

## 6.2 Obstructing Static Analysis

To protect software, Wang *et al.* suggest several transformations at the source code level [4]. Their work presents a method for "control flow graph degeneration", in short *control flow flattening*. To strengthen this model, the use of global pointers is suggested, as under certain conditions may-analysis of pointers can be proven to be NP-hard [9].

Our techniques uses similar insights: we also start from a flattened program, and protect control flow such that extracting control flow information is hard. Rather than relying on global pointers and aliases which can be hard to analyze, we rely on methods which are always hard to analyze. These include inverting the one-wayness of hash functions, the diffusion and bijective properties of both our transition function $F()$ and branching function $B()$.

Secondly, Wang *et al.* do not specify what kind of "complex expressions of subscript calculation" to use. A possible construction proposed by Wang [3] gives as example `swVar = g[g[5]+g[g[23]]]`. However, as an index is not allowed to be out of bound, one can deduce equations from this. This might allow for some reduced brute force attacks. As far as we know no diffusion function is used, which might facilitate constant or range propagation, leading to a set of possible labels, which just have to be checked against the set of hard-coded labels in the switch structure.

Similar to Wang's protection method, our scheme is fully complementary to control flow graph flattening. Furthermore, it can be combined with any other code obfuscation or encryption technique.

### 6.3 Control Flow Flattening

László and Kiss describe in their paper an algorithm to flatten control flow of C++ programs [7]. The work describes methods to translate all control transfer statements (incl. C's jump and switch statements), and a technique to transform error handling constructs (i.e. `try-catch` mechanisms). Some of these transformations seem more challenging than others, forcing the authors to resort to 'tricks' (e.g. the use of `goto`) in certain situations. Furthermore, the authors specify a metric to measure the added complexity of CFG flattening, namely McCabe's cyclomatic number [10]. They conclude that McCabe's complexity metric is increased by a factor 2 to 5 when applying CFG flattening.

When viewed as the complexity of attacking a flattened control flow graph, McCabe's cyclomatic number can be used as a security metric. The formula $v(G) = |E| - |V| + 2$ gives the number of possible execution paths (plus one). Here $|V|$ denotes the number of basic blocks in the CFG, while $|E|$ denotes the number of control transfers. Because of the specific structure of the switch statement, a flattened program (or procedure) always has a fixed structure, see also Figure 1(a). Assume the switch can only transfer control to a node via one edge, then $E \approx 2|V|$ and the cyclomatic number $v(G)$ becomes proportional to $|V|$, the number of basic blocks. We propose to split long basic blocks into pieces increasing $|V|$. Each basic block will also contain less information, necessitating the execution of more blocks, in turn, requiring more work from the attacker.

## 7. CONCLUSION

This paper proposed a method to securely embed control flow information in a program, without statically leaking control flow information. Our solution is based on control flow graph flattening. It extends work of other researchers in a structured and strong way. Our first result forces an adversary to perform global analysis to understand local control transfers. This result holds for both forward and backward analysis. Secondly, we propose three application scenarios which allow to hide control flow behind a secret. This includes a secret key or secret function (in software or hardware), or a construction based on the combination of opaque predicates. To illustrate the strength of our scheme in these scenarios we present the resistance to several attacks. Once one can hide the entire control flow from static analysis, other flow-sensitive internals of the program can also be hidden, e.g. by using aliasing, function pointers, ... Therefore, we see our technique as a contribution to the software protection field.

## 8. ACKNOWLEDGEMENTS

The author Jan Cappaert wishes to acknowledge Jens Hermans and Sebastiaan Indesteege for the informal meetings on discrete logarithms and cryptographic hash functions.

## 9. REFERENCES

[1] C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of 10th ACM Conference on Computer and Communications Security* (CCS 2003), pages 290–299, October 2003.

[2] C. Collberg, C. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. *Principles of Programming Languages 1998* (POPL'98), San Diego, CA, 1998.

[3] C. Wang. *A Security Architecture for Survivability Mechanisms*. PhD thesis, Department of Computer Science, University of Virginia, 2000.

[4] C. Wang, J. Davidson, J. Hill, J. Knight. Protection of Software-based Survivability Mechanisms. *International Conference of Dependable Systems and Networks*, Goteborg, Sweden, July 2001.

[5] M. Dalla Preda, M. Madou, K. De Bosschere and R. Giacobazzi. Opaque Predicate Detection by Abstract Interpretation. In *Proceedings of the 11th International Conference on Algebriac Methodology and Software Technology* (AMAST'06), LNCS, Vol. 4019, pages 81–95, Springer-Verlag, July 2006, Kuressaare, Estonia.

[6] D. Aucsmith. Tamper Resistant Software: an Implementation. In *Proceedings of the First International Workshop on Information Hiding*, LNCS, Vol. 1174, pages 317–333, 1996.

[7] T. László and Á. Kiss. Obfuscating C++ Programs via Control Flow Flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae*, Sectio Computatorica, 30:3–19, August 2009.

[8] J. Ge, S. Chaudhuri, and A. Tyagi. Control Flow Based Obfuscation. In *Proceedings of the 5th ACM workshop on Digital Rights Management*, pages 83–92, Alexandria, VA, USA, 2005.

[9] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems* (LOPLAS), Volume 1, Issue 4, pages 323–337, 1992.

[10] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, p. 308, 1976.

[11] Y. Dodis, A. Elbaz, R. Oliveira and R. Raz. Improved Randomness Extraction from Two Independent Sources. *International Workshop on Randomization and Approximation Techniques in Computer Science* (RANDOM), August 2004.

[12] A. Klimov and A. Shamir. A New Class of Invertible Mappings. *The 4th International Workshop on Cryptographic Hardware and Embedded Systems* (CHES), LNCS, Vol. 2523, pages 470–48, 2002.

[13] W. Blume and R. Eigenmann. Symbolic range propagation. In *Proceedings of the 9th international Symposium on Parallel Processing* (IPPS), pages 357–363, IEEE Computer Society, April 1995.