UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO TÉRCIO PACITTI
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Rafael de Oliveira Costa

TINYOBF: Um Arcabouço de Ofuscação de Código e Proteção de Dados para Rede de Sensores Sem Fio

Rio de Janeiro
2012

Rafael de Oliveira Costa

TinyObf: Um Arcabouço de Ofuscação de Código e Proteção de Dados para Rede de Sensores Sem Fio

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Instituto de Matemática, Instituto Tércio Pacitti, Universidade Federal do Rio de Janeiro, como requisito parcial à obtenção do título de Mestre em Informática

Orientadores: Luci Pirmez, D.Sc
Davidson Rodrigo Boccardo, D.Sc

Rio de Janeiro
2012

C837 Costa, Rafael de Oliveira
TinyObf: Um arcabouço de ofuscação de código e proteção de dados para rede de sensores sem fio. / Rafael de Oliveira Costa. – 2012.
000f.: il.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro, Instituto de Matemática, Instituto Tércio Pacitti, Programa de Pós-Graduação em Informática, Rio de Janeiro, 2012.

Orientadores: Luci Pirmez ; Davidson Rodrigo Boccardo,

1. TinyObf. 2. Segurança de Dados. 3.Rede de Sensores sem Fio – Teses. I. Pirmez, Luci (Orient.). II. Boccardo, Davidson Rodrigo (Orient.). III. Universidade Federal do Rio deJaneiro, Instituto de Matemática, Instituto Tércio Pacitti, Programa de Pós- Graduação em Informática. IV. Título

CDD

Rafael de Oliveira Costa

TINYOBF: Um arcabouço de ofuscação de código e proteção de dados para rede de sensores sem fio

Dissertação de Mestrado submetida ao Corpo Docente do Programa de Pós-Graduação em Informática da Universidade Federal do Rio de Janeiro e à banca externa convidada como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Aprovada em: Rio de Janeiro ___ de _____ de _____.

Prof[a]. Luci Pirmez – Orientadora
D.Sc., COPPE/UFRJ, Brasil

Prof. Davidson Rodrigo Boccardo – Orientador
D.Sc., UNESP, Brasil

Prof. Luiz Fernando Rust da Costa Carmo
Dr., UPS, França

Prof. Luis Henrique Maciel Kosmalski Costa
Dr., UPMC, França

Prof. Raphael Carlos Santos Machado
D.Sc., COPPE/UFRJ, Brasil

Rio de Janeiro

2012

RESUMO

COSTA, Rafael de Oliveira. **TinyObf:** um arcabouço de ofuscação de código e proteção de dados para rede de sensores sem fio. 2012. 000 f. Dissertação (Mestrado em Informática) - Programa de Pós-Graduação em Informática, Instituto de Matemática, Instituto Tércio Pacitti, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2012.

Esse trabalho propõe um arcabouço de ofuscação de código e proteção de dados para Redes de Sensores Sem Fio - *RSSF*. Esse arcabouço, denominado *TinyObf*, foi desenvolvido com o intuito de proteger o programa embarcado nos nós dessa rede contra engenharia reversa. Para isso, esse arcabouço conta com técnicas de ofuscação, capazes de tornar o programa mais difícil de ser analisado, e um mecanismo de proteção de dados que combina tais técnicas com a manipulação do fluxo de controle a fim de criar trechos no programa onde dados sensíveis podem ser escondidos.

O TinyObf pode ser considerado uma solução de segurança adequada para RSSF porque é capaz de proteger a RSSF causando pouca sobrecarga sobre os recursos (processamento, memória e energia) dos sensores. Adicionalmente esse arcabouço pode ser considerado furtivo, pois um programa ofuscado com o TinyObf não fornece indícios de que foi ofuscado, e resistente contra desofuscadores, visto que é difícil desenvolver ferramentas capazes de reverter as ofuscações aplicadas por esse arcabouço.

Palavras-chave: Segurança. Ofuscação de Código. Rede de Sensores Sem Fio.

ABSTRACT

COSTA, Rafael de Oliveira. **TinyObf:** um arcabouço de ofuscação de código e proteção de dados para rede de sensores sem fio. 2012. 000 f. Dissertação (Mestrado em Informática) - Programa de Pós-Graduação em Informática, Instituto de Matemática, Instituto Tércio Pacitti, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2012.

This paper proposes a code obfuscation and data protection framework for Wireless Sensor Networks - WSN. This framework, named TinyObf, was developed in order to protect the embedded software in the WSN nodes against reverse engineering. For this reason, this framework relies on obfuscation techniques, which can make the program more difficult to analyze, and a data protection mechanism, which combines these techniques with control flow manipulation, in order to create program sections where sensitive data can be hidden.

The TinyObf can be considered a suitable security solution for WSN because it can protect the WSN causing little overhead on resources (processing, memory and energy) of the sensors. Additionally this framework can be considered stealthy, because a program obfuscated with TinyObf provides no evidence that was obfuscated, and resistant against deobfuscator, since it is difficult to develop such tool capable to revert the obfuscation performed by this framework.

Keywords: *Security*. *Code Obfuscation*. *Wireless Sensor Networks*.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CF – *Fator de Confusão (Confusion Factor)*

CFG – *Grafo de Fluxo de Controle (Control Flow Graph)*

CG – *Grafo de Chamadas (Call Graph)*

ICFG – *Grafo Interprocedural de Fluxo de Controle (Interprocedural Control Flow Graph)*

MATE – *Man-At-The-End*

MD – *Distância de Mahalanobis (Mahalanobis Distance)*

RSSF – *Rede de Sensores Sem Fio*

Sumário

Os recentes avanços nas tecnologias de sistemas micro-eletromecânicos e nas comunicações sem fio possibilitaram a construção de sensores inteligentes, dispositivos dotados de processador e memória, capazes de monitorar grandezas físicas de um ambiente como, por exemplo, temperatura, luminosidade e ruído, e capaz de comunicar-se com outros sensores através de ondas eletromagnéticas. Devido ao seu tamanho reduzido e baixo custo de produção, dezenas, centenas ou milhares destes sensores têm sido utilizados para criar Redes de Sensores Sem Fio (*RSSFs*), capazes de permitir o desenvolvimento de diversos tipos de aplicações. Por exemplo as RSSFs podem permitir o desenvolvimento de aplicações para o monitoramento de alvos militares, detecção de incidentes ambientais e controle de ambientes inteligentes (YICK *et al.* 2008).

Se por um lado as RSSFs trazem perspectivas para o desenvolvimento de novas aplicações, por outro trazem uma série de desafios. Dentre esses desafios, os principais estão relacionados às vulnerabilidades associadas à comunicação sem fio, à organização *ad-hoc* dos nós dessa rede e a limitação de recursos dos sensores, principalmente em relação a capacidade de energia, pois tais sensores são alimentados por baterias. No entanto, outro desafio é relacionado a segurança das RSSFs, pois os sensores são vulneráveis à captura devido a estarem dispostos em áreas abertas, desprotegidas e às vezes até hostis. Com isso, pessoas mal intencionadas (atacantes) podem ser capazes de capturar tais sensores a fim de realizar um ataque *Man-At-The-End* (*MATE*) com o intuito de comprometer a RSSF.

Um ataque MATE pode ocorrer em qualquer ambiente, desde que o atacante tenha acesso físico ao dispositivo a ser comprometido. Esse tipo de ataque pode ser realizado através da inspeção ou modificação do hardware ou do software embarcado no dispositivo (COLLBERG e NAGRA 2009). Caso um atacante seja capaz de comprometer um nó de uma rede, ele pode ser capaz de comprometer a rede como um todo, pois ele pode além de obter mensagens sensíveis que estejam trafegando, enviar mensagens espúrias nessa rede, fazendo com que a mesma se comporte de maneira inesperada. Portanto esse tipo de ataque pode afetar diversos cenários. Em cenários que utilizam RSSFs, um atacante com acesso físico a um sensor, pode ser capaz de extrair ou adulterar os algoritmos e dados sensíveis do programa embarcado nos sensores (HARTUNG *et al*. 2005). Por exemplo, ao extrair chaves criptográficas, utilizadas para manter a comunicação segura de uma RSSF, um atacante pode ser capaz de decodificar as mensagens que trafegam nessa rede e também transmitir mensagens espúrias. Por outro lado, caso essas chaves sejam utilizadas para codificar dados sensíveis dentro do próprio sensor, e essas chaves sejam descobertas por um atacante, esses dados não estarão mais protegidos, visto que o atacante poderá ser capaz de decodifica-los.

Uma das formas de dificultar um ataque MATE é utilizando algoritmos criptográficos e de compactação. Tais algoritmos têm sido utilizados por desenvolvedores para esconder tanto dados sensíveis como os próprios algoritmos de um programa ao codificar/compactar trechos do programa onde esses elementos estão armazenados. Um problema do uso desses algoritmos é a sobrecarga de processamento gerada por eles, visto que é necessário decodificar/descompactar os trechos protegidos a fim de utilizá-los e codificar/compactar novamente esses trechos após o seu uso. Essa sobrecarga é considerada um problema porque pode prejudicar o funcionamento das aplicações, pois pode afetar o tempo de resposta de uma aplicação. Outro problema do uso desses algoritmos é que eles baseiam sua segurança em um único ponto de falha, ou seja, na chave criptográfica

ou no algoritmo de compactação, pois caso esses elementos não sejam secretos, a segurança desses algoritmos pode ser posta em prova. Outra desvantagem dessas abordagens é que esses algoritmos não são furtivas, pois um código codificado ou compactado pode ser facilmente identificado através da entropia do programa ou pela presença de dados em meio às suas instruções (WARTELL *et al*. 2011); (JEONG *et al*. 2010).

Outra forma de dificultar um ataque MATE é utilizando ofuscação de código. A ofuscação visa produzir um programa semanticamente equivalente ao programa original, mas com alterações em sua sintaxe capazes de dificultar a engenharia reversa, ou seja, processo capaz de reconstruir o código executável de um programa em uma estrutura de mais alto nível com o intuito de identificar os componentes desse programa e as suas relações a fim de obter algum conhecimento. A engenharia reversa é comumente dividida em duas fases: desmontagem (*disassembly*) e descompilação. Na fase de desmontagem, o código binário é traduzido para código assembly, ou seja, uma representação dos códigos de máquina que devem ser executadas pelo processador. Na fase de descompilação, o código assembly, gerado pela fase anterior, é utilizado para gerar estruturas em uma linguagem de mais alto nível, a fim de facilitar a análise e extração de conhecimento do programa. Portanto caso um atacante esteja disposto a extrair informações ou modificar um programa, ele deverá realizar engenharia reversa. Por exemplo, um atacante com o intuito de usufruir gratuitamente de um programa protegido com mecanismos que requerem uma licença de utilização, deverá realizar a engenharia reversa a fim de identificar tais mecanismos, para em seguida removê-los a fim de poder usufruir do programa sem a necessidade de comprar a sua licença de utilização.

A ofuscação pode ser utilizada para proteger um programa contra engenharia reversa porque as transformações realizadas no seu código podem comprometer tanto a fase de desmontagem como a de descompilação realizadas pelas ferramentas de engenharia reversa. Na literatura, a maioria dos trabalhos de ofuscação concentra seus esforços na criação de técnicas capazes de comprometer a fase de descompilação (WANG 2000), (CHO *et al*. 2001), (OGISO *et al*. 2003). No entanto, outras propostas apresentam técnicas de ofuscação capazes de comprometer a desmontagem de um programa (LINN e DEBRAY 2003).

As técnicas de ofuscação capazes de comprometer a desmontagem buscam corromper as premissas assumidas pelas ferramentas de engenharia reversa. Por exemplo, essas técnicas modificam a sequência de instruções utilizadas para realizar uma chamada ou retorno de função a fim de induzir com que as ferramentas de engenharia reversa traduzam de forma incorreta o código binário. Uma vantagem desse tipo de ofuscação é que é difícil desenvolver ferramentas (desofuscadores) capazes de reverter esse tipo de ofuscação, pois é difícil identificar os pontos do programa que foram ofuscados já que as transformações realizadas pela ofuscação podem ser aleatórias desde que mantenha o funcionamento correto do programa. Outra vantagem desse tipo de ofuscação é que a sua utilização não sobrecarrega os recursos computacionais (processamento e memória) de forma negativa porque esse tipo de ofuscação simplesmente substitui ou adiciona instruções ao programa, aumentando minimamente o número de ciclos de processamento e o tamanho do programa.

Uma desvantagem da ofuscação que compromete a fase de desmontagem é que esse tipo de ofuscação pode modificar a distribuição de instruções, o número de laços e o uso de operandos, tornando o programa ofuscado muito diferente de um programa comum não ofuscado. Do ponto de vista da distribuição de instruções, a ofuscação pode causar anomalias em relação à ocorrência de determinadas instruções, permitindo com que seja possível identificar pontos do programa que foram ofuscados. Dessa forma, é necessário eliminar tais anomalias

a fim de tornar o programa ofuscado furtivo, ou seja, capaz de não dar indícios de que foi ofuscado.

Apesar da necessidade de proteger a RSSF contra ataques MATE, é necessário observar o tempo de vida da RSSF antes de desenvolver uma solução de segurança para esse tipo de rede. Isso se deve porque, em geral, os sensores são alimentados por baterias não recarregáveis, e portanto o tempo de vida da RSSF é limitado pela capacidade da bateria. Tal limitação poderia ser solucionada, caso as baterias pudessem ser substituídas. No entanto não é comum a elaboração de uma política de substituição dessas baterias, por conta da localização desses sensores. Portanto é necessário desenvolver uma solução capaz de economizar o máximo de energia possível.

1.1 Objetivos

O objetivo desse trabalho é propor um arcabouço de ofuscação de código e proteção de dados eficaz, eficiente e furtivo, capaz de proteger uma RSSF contra ataques MATE. Esse arcabouço é eficaz porque utiliza técnicas de ofuscação capazes de dificultar a engenharia reversa, tornando mais difícil extrair ou adulterar algoritmos e dados sensíveis embarcado nos nós da RSSF; é eficiente porque tais técnicas não sobrecarregam demasiadamente os recursos dos sensores (tamanho de código, ciclos de processamento e consumo de energia), impactando minimamente no tempo de vida da RSSF; e é furtivo porque um programa ofuscado com esse arcabouço não gera indícios de que foi ofuscado. Adicionalmente, esse arcabouço pode ser considerado resistente contra desofuscadores, visto que é difícil desenvolver funções capazes de detectar as instruções relacionadas com a ofuscação, pois esse arcabouço dispõe tais instruções de forma aleatória dentro do código do programa.

Uma das vantagens dessa proposta é que a arquitetura desse arcabouço pode ser personalizada a fim de proteger RSSF cujos nós são dispositivos de diferentes plataformas de hardware. Para isso, basta que as técnicas de ofuscação sejam personalizadas de acordo com conjunto de instrução da plataforma de hardware do sensor desejado. Outra vantagem desse arcabouço é que as técnicas de ofuscações podem ser aplicadas em função do perfil de segurança, ou seja, um valor que define o uso da ofuscação considerando a segurança e outros requisitos das RSSFs, pois existem situações onde é mais importante garantir o funcionamento da RSSF do que manter o nível de segurança.

Nesse arcabouço, escolhemos utilizar somente técnicas de ofuscação que comprometem a desmontagem de um programa porque acreditamos que a aplicação desse tipo de ofuscação é suficiente para desmotivar um atacante a realizar a engenharia reversa e porque esse tipo de técnica gera uma sobrecarga pequena em relação aos os recursos dos sensores.

Assumindo que um programa geralmente é constituído de um segmento de código e um segmento de dados, o mecanismo de proteção de dados proposto nesse arcabouço combina as técnicas de ofuscação com a manipulação do fluxo de controle a fim de criar trechos no segmento de código que nunca serão executados. Com isso, dados sensíveis podem ser armazenados nesses trechos, induzindo com que as ferramentas de engenharia reversa traduzam esses dados, como se fossem instruções do programa. Portanto localizar dados em um programa ofuscado com nosso arcabouço, é uma tarefa mais difícil, visto que tais dados sensíveis estarão camuflados em meio as demais instruções do programa.

Como prova de conceito, nós desenvolvemos o TinyObf, uma personalização do arcabouço proposto

para a arquitetura de hardware Atmel AVR, utilizada por sensores da plataforma MICAz. Nessa prova de conceito, manipulamos o fluxo de controle dos programas ofuscados com o TinyObf com o intuito de proteger uma chave criptográfica.

Em suma, esse trabalho busca desenvolver uma solução de segurança capaz de proteger as RSSFs contra ataques MATE, dificultando a extração e adulteração de dados sensíveis e algoritmos através da engenharia reversa. As contribuições desse trabalho são:

(i) Proposta de uma arquitetura lógica de um arcabouço de ofuscação de código e proteção de dados eficaz, eficiente e furtivo que pode ser personalizado para diferentes arquiteturas de hardware;

(ii) Proposta de um mecanismo de proteção de dados capaz de camuflar dados sensíveis de um programa em meio as demais instruções do programa;

(iii) Personalização das técnicas de ofuscação existentes na literatura para a arquitetura de processadores Atmel AVR.

1.2 Organização

O restante deste trabalho está organizado em sete seções. Na seção 0 apresenta-se os conceitos básicos de engenharia reversa, ofuscação de código e segurança em RSSF*s*. Na seção 0 são destacadas as diferenças e similaridades de trabalhos encontrados na literatura com o presente trabalho. Na seção 0, a arquitetura do *TinyObf* é apresentada assim como seu funcionamento. Na seção 0 apresenta-se um estudo de caso mostrando a personalização do arcabouço proposto para a arquitetura *Atmel AVR* e como as técnicas de ofuscação desse arcabouço podem ser utilizadas para proteger dados confidenciais. Na seção 0 são expostos como os experimentos com o arcabouço foram realizados e os resultados obtidos. Por último, na seção 0 são apresentadas as conclusões e os trabalhos futuros.

Conceitos Básicos

Neste capítulo serão apresentados conceitos básicos de engenharia reversa, ofuscação de código e segurança em RSSFs a fim de servir como base para o entendimento desse trabalho.

1.3 Engenharia Reversa

Geralmente os programas são distribuídos para os usuários finais através do seu código binário, sem que o respectivo código fonte seja disponibilizado. Dessa forma, entender o comportamento interno de um programa é uma tarefa muito difícil de ser realizada. No entanto existem situações que requerem este conhecimento, seja para reparar alguma falha de funcionamento ou para adicionar novas funcionalidades ao programa. Nessas situações, é necessário realizar engenharia reversa a fim de tentar identificar os componentes do programa e a relação entre eles.

A engenharia reversa de um programa é um processo que tenta reconstruir o código binário de um programa em uma estrutura de mais alto nível a fim de facilitar a análise de um programa. No entanto, apesar dos benefícios que a engenharia reversa pode oferecer, existem situações em que ela é utilizada com intuitos

maliciosos. Nesses casos, a engenharia reversa é realizada para extrair dados confidenciais ou algoritmos proprietários de um programa.

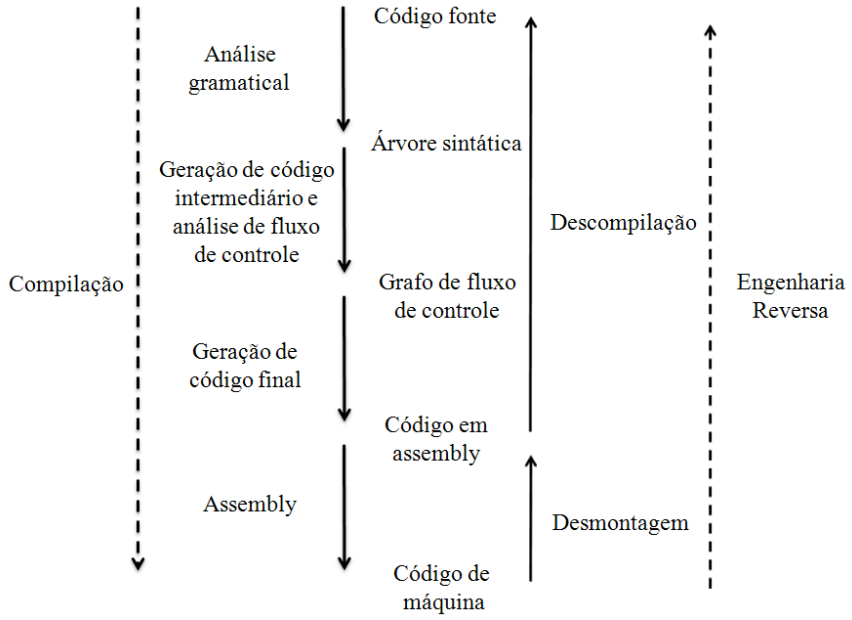

Figura 1. Fases de compilação e de engenharia reversa

A engenharia reversa pode ser vista como sendo o caminho inverso da compilação e comumente é dividida em duas fases: a fase de desmontagem (*disassembly*) e a fase de descompilação, como mostrado na Figura 1. Durante a fase de desmontagem os códigos de máquina de um código binário são traduzidos para um código em *assembly*, notação utilizada para representar as instruções que deverão ser executadas por um processador. Em seguida, na fase de descompilação, estruturas de alto nível são construídas a fim de facilitar a análise de um programa, já que analisar um programa a partir de um código em assembly é uma tarefa que requer muito tempo e esforço.

A descompilação é um processo totalmente dependente da desmontagem, pois precisa do código assembly gerado na fase anterior (desmontagem) para gerar um código fonte que facilite o entendimento do programa. Portanto qualquer erro durante a fase de desmontagem de um programa pode causar erros na fase de descompilação.

Nas próximas subseções serão apresentados os principais algoritmos de desmontagem e como é realizada a análise de um programa. Nesse trabalho somente detalhamos os algoritmos de desmontagem porque nossa proposta tem como objetivo utilizar técnicas de ofuscação que comprometem esse tipo de algoritmo.

Algoritmos de Desmontagem

Atualmente os dois principais algoritmos de desmontagem utilizados pelas ferramentas de engenharia reversa são: varredura linear e transversal recursivo.

A Figura 2 apresenta o algoritmo varredura linear. Nesta figura, a função principal (main) define o endereço do primeiro byte (*endInicial*) e do último byte (*endFinal*) do código do programa antes de executar a função *varreduraLinear* (linhas 12-17). A função *varreduraLinear*, iniciada na linha 4, verifica se o endereço *e* está contido entre os endereços *endInicial* e *endFinal* (linha 6) e, enquanto essa condição for verdadeira, o

algoritmo tenta fazer a tradução dos bytes a partir desse endereço, e continua de forma linear a partir do próximo endereço (linhas 7 e 8). Essa tradução é feita verificando se os bytes armazenados a partir do endereço *e* constituem o código de uma instrução válida do conjunto de instruções em questão, específico para a arquitetura de hardware que esse programa foi compilado.

A Figura 3, por sua vez, apresenta o algoritmo transversal recursivo. Esse algoritmo assim como o algoritmo varredura linear inicia a desmontagem de um programa a partir do primeiro byte do código binário do programa (linha 24). A desmontagem é realizada pela função *transversalRecursivo* (linha 4), que após verificar se o endereço *e* está contido entre os endereços *endInicial* e *endFinal* (linha 6), marca-o como um endereço visitado e tenta traduzir os bytes a partir desse endereço (linhas 9 e 10). Caso a instrução *I* seja uma instrução que redireciona o fluxo de controle, tal como uma instrução *JMP*, a desmontagem não irá seguir sequencialmente. Nesse caso, o algoritmo tentará identificar o endereço de destino dessa instrução e tentará continuar o processo de tradução a partir desse endereço (linhas 11-14). O processo de tradução só não ocorre dessa maneira caso o endereço *e* já tenha sido marcado, pois isso significa que esse endereço já foi visitado (linhas 7 e 8).

```
1    var endInicial
2    var endFinal
3
4    proc varreduraLinear( e )
5    begin
6        while( endInicial<= e <endFinal ) do
7            I := traduz instrução no endereço e;
8            e += tamanho( I );
9        done
10  end
11
12   proc main( )
13   begin
14       endInicial := endereço do primeiro byte do código executável;
15       endFinal := endInicial + tamanho do código executável;
16       varreduraLinear( endInicial );
17   end
```

Figura 2 Algoritmo varredura linear.

Independente do algoritmo de desmontagem utilizado, alguns aspectos do código são identificados de acordo com algumas premissas utilizadas pelos *disassemblers*, ou seja, ferramentas que realizam a desmontagem. Por exemplo na arquitetura x86, uma das premissas assumida pelos disassemblers é que o endereço de retorno de uma função é o endereço localizado após o endereço da instrução utilizada para realizar a chamada de uma função, geralmente instrução 'CALL'. No entanto, tais premissas podem não ser obedecidas durante a compilação por conta da otimização ou ofuscação desse código. Dessa forma a desmontagem do programa pode ser questionável.

Outra premissa assumida pelos disassemblers é que os dados do programa são armazenados no segmento de dados. Portanto ao armazenar dados no segmento de código, tais dados poderão ser considerados

como se fossem instruções. Para isso é necessário garantir que o endereço do segmento de código onde esses dados estão armazenados nunca serão executados e, além disso, modificar todas as referencias para esses dados dentro do código.

```
1    var endInicial
2    var endFinal
3
4    proc transversalRecursivo(e)
5    begin
6        while( endInicial<= e < endFinal ) do
7            if ( endereço e já foi visitado )
8                return;
9            marcar e como um endereço visitado;
10           I := traduz instrução no endereço e;
11           if ( I é uma chamada de função or I é um salto condicional )
12               for each possível alvo t de I do
13                   transversalRecursivo( t );
14               done
15           else
16               e += tamanho( I );
17       done
18   end
19
20   proc main()
21   begin
22       endInicial := endereço do primeiro byte do código executável;
23       endFinal := endInicial + tamanho do segmento de código;
24       transversalRecursivo( endInicial );
25   end
```

Figura 3 Algoritmo transversal recursivo

Análise de Programas

A análise de programas pode ser classificada em relação ao (i) fluxo de controle e ao (ii) fluxo de dados. A análise de fluxo de controle considera a sequencialidade de instruções executadas, descrevendo, portanto, sua lógica de execução. Já a analise de fluxo de dados é relacionada a um processo de descoberta do comportamento dos dados de um programa durante a sua execução. Esta análise depende das estruturas criadas pela primeira, então resultados incorretos ou imprecisos durante a análise de fluxo de controle certamente irão impactar na precisão da análise de fluxo de dados. Ambas as análises podem ser realizadas tanto intraproceduralmente quanto interproceduralmente. Na análise intraprocedural, as funções do programa são analisadas individualmente, enquanto na análise interprocedural busca-se entender também o relacionamento entre todas as funções (SHARIR e PNUELI 1981).

A análise de fluxo de controle é comumente feita utilizando o grafo de fluxo de controle

interprocedural[1] (*ICFG*). Um *ICFG* é representado pela composição de um grafo de chamadas[2] (*CG*) com os grafos de fluxo de controle[3] (*CFGs*) de um programa. Um *CG* é um multigrafo direcionado cujos nós representam as funções e suas arestas representam as chamadas de função do programa. Um *CFG* representa os possíveis fluxos de uma função e é uma representação conservativa, pois apresenta o conjunto de todos possíveis caminhos do fluxo de controle. Cada nó (bloco básico) de um *CFG* consiste em um conjunto de instruções a serem executadas e cada aresta, localizada no fim de um bloco básico é representada por uma instrução de salto condicional ou incondicional que redireciona o fluxo do programa. Cada *CFG* possui dois tipos especiais de nós: nó inicial[4] e nó final[5]. O nó inicial representa o ponto onde a função inicia e o nó final representa o ponto onde a função termina. Cada chamada de função no programa, ou seja, cada vértice no CG é representado no *ICFG* por dois nós: nó de chamada[6] e nó de retorno[7]. Um nó de chamada conecta o endereço da chamada de função com o nó inicial da função que será executada. Um nó de retorno, por sua vez, conecta o endereço de retorno da chamada de função com o nó final da função.

Em seguida será mostrado como é representado um ICFG de um programa. Para isso foi criado um código exemplo, o qual será utilizado para formar um ICFG. Dessa forma, a Figura 4 mostra um programa exemplo que simplesmente é capaz de realizar a soma dois números. Neste programa existem duas funções: Main e Add. Na função Main antes de cada chamada de função para a função Add, são inseridos na pilha os valores a serem somados. Na função Add é onde realmente esses números, localizados na pilha, são somados. Dessa forma, pode-se observar que as chamadas de função para a função Add estão localizadas em A3 e A6 dentro da função Main enquanto os endereços de retorno relacionados com a função Add estão localizados em A4 e A7.

O ICFG apresentado na Figura 5 é composto por dois CFGs, o CFG da função Main e o CFG da função Add. O nós inicial e final do CFG da função Main são denominados $Inicial_{Main}$ e $Final_{Main}$ e os nós inicial e final do CFG da função Add são denominados $Inicial_{Add}$ e $Final_{Add}$.

```
Main:
        A1:   PUSH 0x1
        A2:   PUSH 0x2
        A3:   CALL Add
        A4:   PUSH 0x3
        A5:   PUSH 0x4
        A6:   CALL Add
        A7:   JMP A9
        A8:   NOP
        A9:   RET
```

1 Original do inglês, interprocedural *control flow graph*
2 Original do inglês, *call graph*
3 Original do inglês, *contr*ol *flow graphs*
4 Original do inglês, *entry node*
5 Original do inglês, *exit node*
6 Original do inglês, *call node*
7 Original do inglês, *return node*

```
Add:
        A10:  MOV eax,[esp+4]
        A11:  MOV ebx,[esp+8]
        A12:  ADD eax,ebx
        A13:  RET
```

Figura 4 Programa que utiliza uma função de dois números

Dentro de cada CFG mostrado na Figura 5 existem arestas intraprocedurais que indicam a lógica de execução da própria função e arestas interprocedurais que mostra a relação entre as duas funções (Main e Add). A primeira chamada de função (linha A3) é representada no ICFG pelos nós de chamada e retorno localizados nos endereços A3.1 e A3.2 respectivamente (Figura 5). A segunda chamada de função (linha A6) é representada no ICFG pelos nós de chamada e retorno localizados nos endereços A6.1 e A6.2 respectivamente (Figura 5). Note que as arestas interprocedurais de chamada sempre ligam cada nó de chamada com o nó inicial do CFG da função que está sendo chamada, enquanto as arestas interprocedurais de retorno ligam o nó final da função que foi chamada com o nó de retorno da instrução que realizou a chamada de função.

Em uma análise intraprocedural, as conclusões são obtidas através do funcionamento de cada função, sem considerar os relacionamentos entre essas funções, gerando resultados com menor precisão. Essa análise considera as instruções que serão executadas dentro de um bloco básico e por isso não dá uma noção geral sobre o comportamento do programa. Em contrapartida, em uma análise interprocedural, o fluxo de execução entre as funções é considerado, permitindo entender melhor o comportamento do programa como um todo. No entanto, caso haja problemas durante a identificação dos nós de chamada e de retorno de uma função, a análise do programa pode ser comprometida, podendo induzir um atacante a obter resultados não confiáveis de sua análise, pois uma vez que os limites de uma função não foram corretamente definidos, pode-se ter uma visão incorreta das instruções que fazem parte de uma determinada função, gerando erro na compreensão do comportamento de tal função.

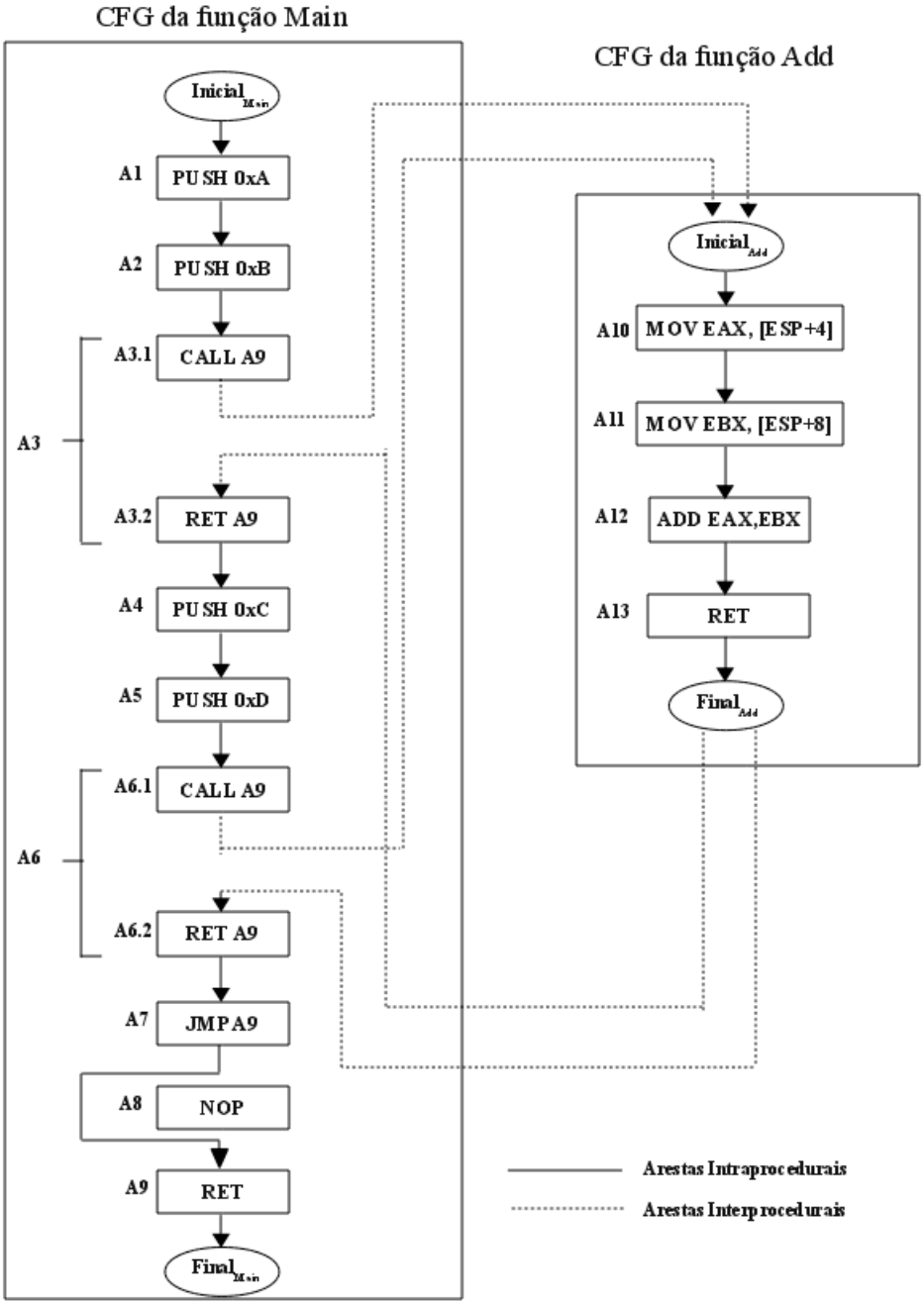

Figura 5 ICFG do programa original apresentado na Figura 4

1.4 Ofuscação de Código

A ofuscação de código é uma técnica que visa produzir um programa semanticamente equivalente ao programa original, mas com alterações em sua sintaxe, capazes de dificultar o entendimento desse programa. A ofuscação é utilizada para dificultar a engenharia reversa de um programa e com isso tentar impedir a extração ou adulteração de dados confidenciais ou de algoritmos de um programa.

Em alguns casos, a ofuscação é usada para diversificar um programa, ou seja, criar versões diferentes de um mesmo programa com o intuito de combater ataques de colisão, no qual um atacante com posse de mais de uma cópia desse programa, consegue descobrir a localização de uma determinada propriedade do programa, como por exemplo, *fingerprint*, chave criptográfica ou marca d'água. Ao aplicar ofuscação de código, um ataque

de colisão não é capaz de identificar padrões entre diferentes versões do programa, dificultando com que suas propriedades sejam descobertas (BOCCARDO *et al*. 2010) já que a forma de cada versão é diferente uma da outra apesar de terem o mesmo comportamento.

A ofuscação de código pode ser aplicada tanto para comprometer a fase de desmontagem como a fase de descompilação da engenharia reversa. A maioria das técnicas de ofuscação atua na fase de descompilação, dificultando a geração de estruturas de alto nível. Determinadas técnicas de ofuscação buscam alterar o fluxo de controle do programa, como por exemplo, a junção[8] e separação[9] de funções (BALAKRISHNAN e SHULZE 2005). Esse tipo de técnica compromete a delimitação das funções de um programa. Ao dificultar a identificação do início ou fim de uma função, a análise se torna mais difícil, pois um atacante busca compreender um programa a partir de informações obtidas da análise de suas funções.

Outro exemplo de técnica de ofuscação que atua na fase de descompilação é a reordenação ou inclusão de blocos básicos (vértices) e de relacionamentos entre esses blocos (arestas) do *CFG* a fim de alterar o fluxo de controle do programa (BALAKRISHNAN e SHULZE 2005). Além disso, As técnicas de ofuscação também podem ser aplicadas para dificultar a análise do fluxo de dados e não somente o fluxo de controle, como por exemplo, substituindo expressões simples por outras mais complexas (BOCCARDO *et al*. 2010).

As técnicas de ofuscação que comprometem a fase de desmontagem, por sua vez, baseiam-se na fragilidade das convenções de compilação a fim de induzir com que os disassemblers traduzam incorretamente as instruções de um programa. Essas técnicas são realizadas através da substituição e inserção de instruções. Dentre essas técnicas, destacamos a ofuscação de chamada, ofuscação de retorno (LAKHOTIA *et al*. 2010), ofuscação de falso retorno (VIEGA *et al*. 2003) e a ofuscação de salto incondicional (LINN e DEBRAY 2003) que serão apresentadas com mais detalhes nas seções seguintes.

Ofuscação de Chamada

De acordo com as convenções de compilação, os disassemblers identificam o início e o fim de uma função através de instruções de chamada e de retorno. O destino de uma instrução de chamada identifica o início de uma função e a instrução de retorno identifica o fim dessa função.

A ofuscação de chamada é uma técnica capaz de remover a delimitação de funções ao remover as instruções de chamada e, consequentemente, a identificação do início de uma função de um programa. Portanto, qualquer análise interprocedural realizada em um programa ofuscado com essa técnica será comprometida, pois a mesma depende da delimitação de funções para gerar o *ICFG*. Caso todas as chamadas de função de um programa sejam ofuscadas, as ferramentas de engenharia reversa identificarão o programa como um conjunto de instruções contidas em uma única função. Esse tipo de ofuscação dificulta a análise de um programa visto que é mais fácil para um atacante analisar várias funções menores do que analisar uma única função que contempla toda a execução do programa.

A ofuscação de chamada pode ser implementada substituindo as instruções de chamada de função, geralmente instruções *CALL*, por um conjunto de instruções capazes de manter a mesma semântica dessa instrução. Geralmente uma instrução *CALL* realiza as seguintes operações: (i) armazena o endereço de retorno da

[8] Original do Inglês, *inlining*
[9] Original do inglês, *outlining*

função no topo da pilha e (ii) redireciona o fluxo de controle para o endereço da função a ser executada. Na arquitetura x86, podemos substituir uma instrução *CALL* por duas instruções *PUSH* e uma instrução *RET*. A primeira instrução *PUSH* é utilizada para armazenar no topo da pilha o endereço de retorno da função, endereço para onde o fluxo de controle deverá ser redirecionado após o fim da execução da função chamada. A segunda instrução *PUSH* é utilizada para armazenar no topo da pilha o endereço da função a ser executada. Por último, a instrução *RET* é utilizada a fim de redirecionar o fluxo de controle para o endereço que está no topo da pilha, ou seja, o endereço da função a ser executada.

A Figura 6 mostra um exemplo onde foi utilizado a ofuscação de chamada. A coluna da esquerda (Figura 6(a)) mostra um programa baseado no programa apresentado na Figura 4. Na Figura 6(b) é mostrado como as chamadas de função localizadas em B3 e B6 foram ofuscadas. As instruções localizadas entre B3-B5 e B8-B10 da Figura 6(b) representam as chamadas de função subsituídas. A instrução "PUSH B6" (B3) empilha o endereço de retorno da função e a instrução "PUSH B14" (B4) empilha o endereço da função a ser executada. Similarmente as instruções "PUSH B11" (B8) e "PUSH B14" (B9) também empilham o endereço de retorno e o endereço da função que será executada. Quando as instruções RET localizadas em B5 e B10 forem executadas, o fluxo de controle será redirecionado para o endereço armazenado no topo da pilha, ou seja, o endereço da função *Add* (B14).

```
Main:
      B1:   PUSH 0xA
      B2:   PUSH 0xB
      B3:   CALL Add
      B4:   PUSH 0xC
      B5:   PUSH 0xD
      B6:   CALL Add
      B7:   JMP B9
      B8:   NOP
      B9:   RET
Add:
      B10:  MOV eax,[esp+4]
      B11:  MOV ebx,[esp+8]
      B12:  ADD eax,ebx
      B13:  RET
```

(a)

```
Main:
      B1:   PUSH 0xA
      B2:   PUSH 0xB
      B3:   PUSH B6
      B4:   PUSH B14
      B5:   RET
      B6:   PUSH 0xC
      B7:   PUSH 0xD
      B8:   PUSH B11
      B9:   PUSH B14
      B10:  RET
      B11:  JMP B13
      B12:  NOP
      B13:  RET
Add:
      B14:  MOV eax,[esp+4]
      B15:  MOV ebx,[esp+8]
      B16:  ADD eax,ebx
      B17:  RET
```

(b)

Figura 6 Programa que exemplifica o uso da ofuscação de chamada

A Figura 7 representa o *ICFG* do programa ofuscado apresentado na Figura 6(b). Neste ICFG pode-se notar que as arestas interprocedurais foram removidas devido as chamadas de função terem sido ofuscadas. Com isso não é possível ver que existe relação entre a função Main e a função Add. Por outro lado, devido a inserção

da instrução RET ao substituir a instrução CALL, a delimitação da função Main foi prejudicada, pois nos endereços onde estão essas instruções (B5 e B10) foi considerado como fim de funções. Dessa forma existe arestas entre esses endereços (B5 e B10) e $Final_{Main}$.

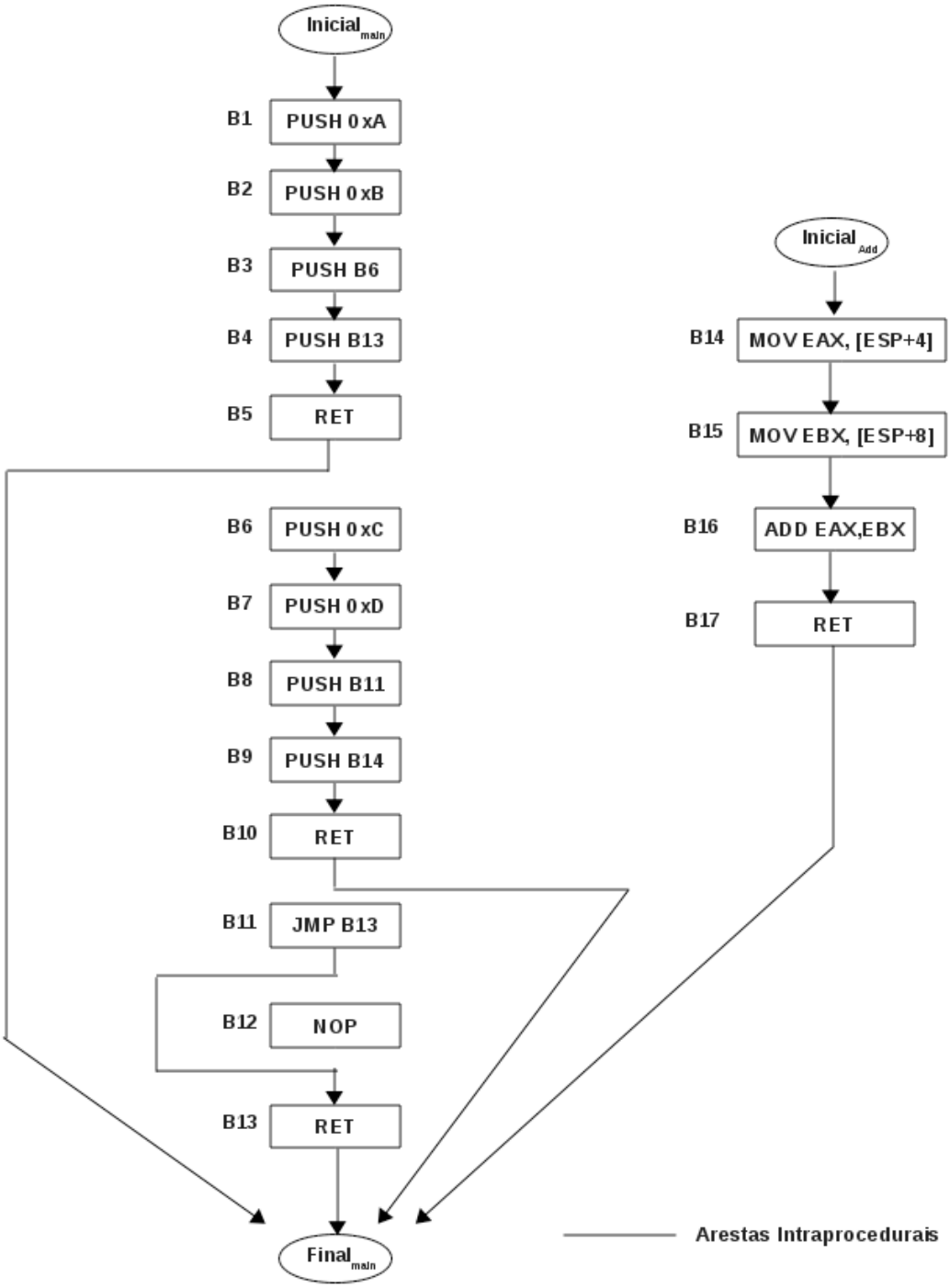

Figura 7. ICFG gerado a partir do programa ofuscado usando a ofuscação de chamada

Ofuscação de Retorno

A ofuscação de retorno é uma técnica de ofuscação que visa remover as instruções de retorno de um programa com o intuito de dificultar a identificação do término de uma função. Esse tipo de ofuscação dificulta a análise interprocedural de um programa, pois a mesma depende da delimitação do fim das funções para gerar o *ICFG*. Quando a ofuscação de retorno é utilizada, o fim de uma função passa a ser o fim de uma outra função, representado pela próxima instrução RET. Com isso, a análise intraprocedural da função ofuscada retornará resultados menos precisos do que se fosse realizada a análise no grafo legítimo.

A ofuscação de retorno pode ser implementada substituindo as instruções de retorno por outras instruções que também atuam como uma instrução de retorno. Uma instrução de retorno redireciona o fluxo de controle para o endereço armazenado no topo da pilha. Uma ofuscação de retorno em uma arquitetura x86 pode ser realizada substituindo uma instrução *RET* por uma instrução *POP* seguida de uma instrução *JMP*. A instrução *POP* é utilizada para remover o endereço de retorno do topo da pilha e armazená-lo em um registrador. E a instrução JMP redireciona o fluxo de controle do programa para o endereço armazenado no registrador utilizado pela instrução *POP*.

```
Main:                              Main:
      C1:   PUSH 0x1                     C1:   PUSH 0xA
      C2:   PUSH 0x2                     C2:   PUSH 0xB
      C3:   CALL Add                     C3:   CALL Add
      C4:   PUSH 0xC                     C4:   PUSH 0xC
      C5:   PUSH 0xD                     C5:   PUSH 0xD
      C6:   CALL Add                     C6:   CALL Add
      C7:   JMP B9                       C7:   JMP C9
      C8:   NOP                          C8:   NOP
      C9:   RET                          C9:   POP ECX
Add:                                     C10:  JMP ECX
      C10:  MOV eax,[esp+4]        Add:
      C11:  MOV ebx,[esp+8]              C11:  MOV eax,[esp+4]
      C12:  ADD eax,ebx                  C12:  MOV ebx,[esp+8]
      C13:  RET                          C13:  ADD eax,ebx
                                         C14:  POP ECX
                                         C15:  JMP ECX

            (a)                          (b)
```

Figura 8. Programa que exemplifica o uso da ofuscação de retorno

A Figura 8 mostra um programa ofuscado usando ofuscação de retorno. A coluna à esquerda mostra o programa original (Figura 8(a)). Na coluna da direita (Figura 8(b)) mostra como foi realizado a substituição das instruções de retorno. As instruções de retorno (C9 e C13) foram substituídas pelas instruções "POP ECX" e "JMP ECX" como pode ser visto nas linhas C9-C10 e C14-C15. Com isso o *ICFG* foi corrompido já que não foi possível criar as arestas interprocedurais de retorno, ou seja, as arestas partindo do fim da função Add para cada retorno de função (C3.2 e C6.2) e isso se deve porque essas ferramentas não são capazes de identificar o final da

função Add.

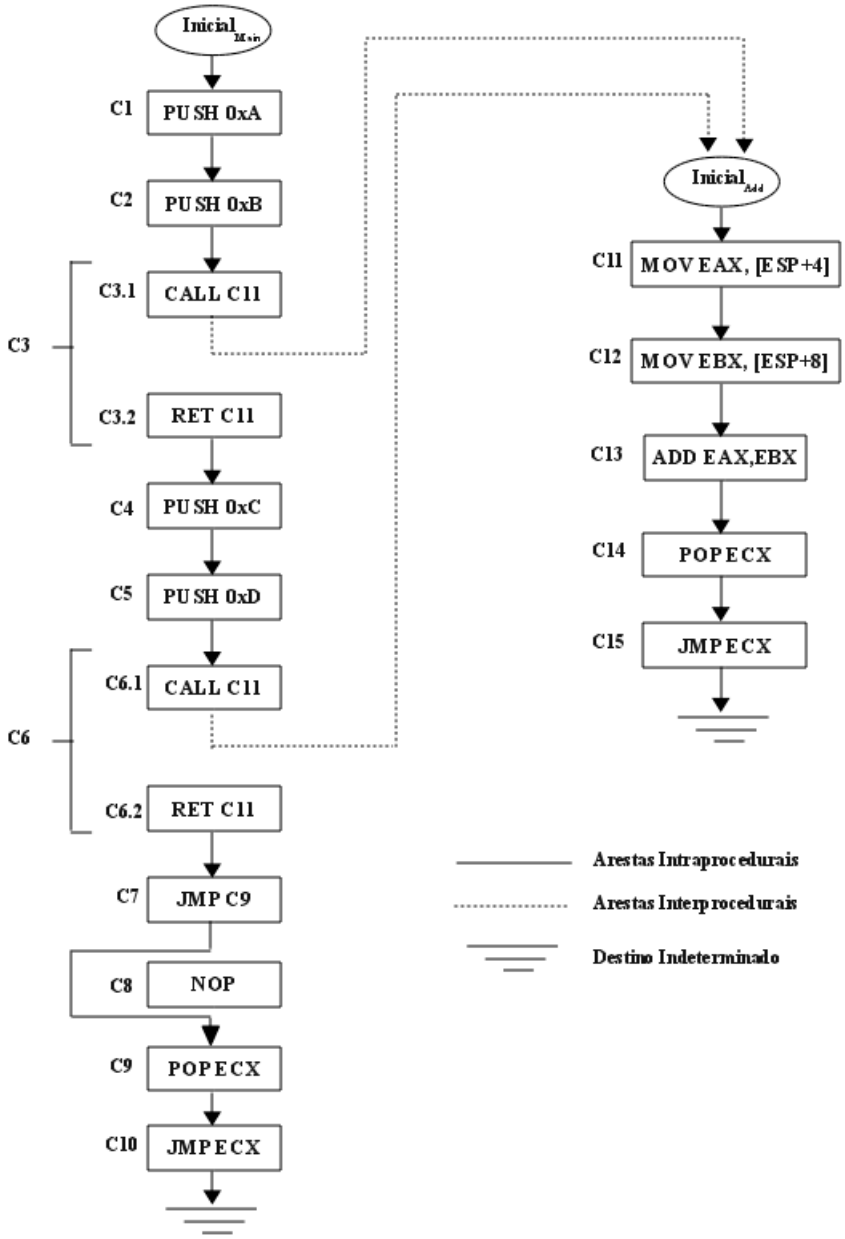

Figura 9. ICFG gerado após a aplicação da ofuscação de retorno.

Ofuscação de Falso Retorno

A ofuscação de falso retorno é uma técnica de ofuscação que visa inserir um falso retorno (instrução de retorno que não caracteriza o fim legítimo de uma função) no meio de uma função a fim de comprometer a delimitação dessa função. Diferentemente da ofuscação de retorno que remove a instrução de retorno, na ofuscação de falso retorno, uma instrução de retorno é inserida a fim de induzir que as ferramentas de engenharia reversa identifiquem prematuramente o fim de uma função.

A implementação de uma ofuscação de falso retorno depende da manipulação do fluxo de controle do programa, pois é necessário preservar a semântica da função onde a instrução de retorno foi inserida a fim de

garantir que o fluxo de controle seja redirecionado para a próxima instrução a ser executada. Na arquitetura x86, esse tipo de ofuscação pode ser realizada inserindo uma instrução PUSH e uma instrução RET dentro da função. A instrução PUSH fica responsável por adicionar no topo da pilha o endereço da próxima instrução a ser executada enquanto a instrução RET atuará como o falso retorno já que essa instrução na verdade redirecionará o fluxo de controle para o endereço armazenado no topo da pilha, ou seja, o endereço da próxima instrução a ser executada.

```
Main:
      D1:   PUSH 0x1
      D2:   PUSH 0x2
      D3:   CALL Add
      D4:   PUSH 0xC
      D5:   PUSH 0xD
      D6:   CALL Add
      D7:   JMP B9
      D8:   NOP
      D9:   RET
Add:
      D10:  MOV eax,[esp+4]
      D11:  MOV ebx,[esp+8]
      D12:  ADD eax,ebx
      D13:  RET
```

(a)

```
Main:
      D1:   PUSH 0xA
      D2:   PUSH 0xB
      D3:   CALL Add
      D4:   PUSH 0xC
      D5:   PUSH 0xD
      D6:   CALL Add
      D7:   JMP C9
      D8:   NOP
      D9:   RET
Add:
      D10:  MOV eax,[esp+4]
      D11:  MOV ebx,[esp+8]
      D12:  PUSH D14
      D13:  RET
      D14:  ADD eax,ebx
      D15:  RET
```

(b)

Figura 10. Programa que exemplifica o uso da ofuscação de falso retorno

A Figura 10(b) mostra um exemplo onde foi inserido um falso retorno na função Add. A instrução "RET" em D13 atua como falso retorno induzindo a identificação incorreta do fim da função Add, quando na verdade essa instrução redireciona o fluxo de controle para a próxima instrução a ser executada dentro da própria função Add, ou seja, 'ADD eax,ebx'. Neste exemplo, a instrução "PUSH D14" (D12) é utilizada para empilhar o endereço de retorno utilizado pelo falso retorno (D14) a fim de garantir o mesmo comportamento da função Add já que a próxima instrução a ser executada é a instrução 'ADD eax, ebx' localizada em D14.

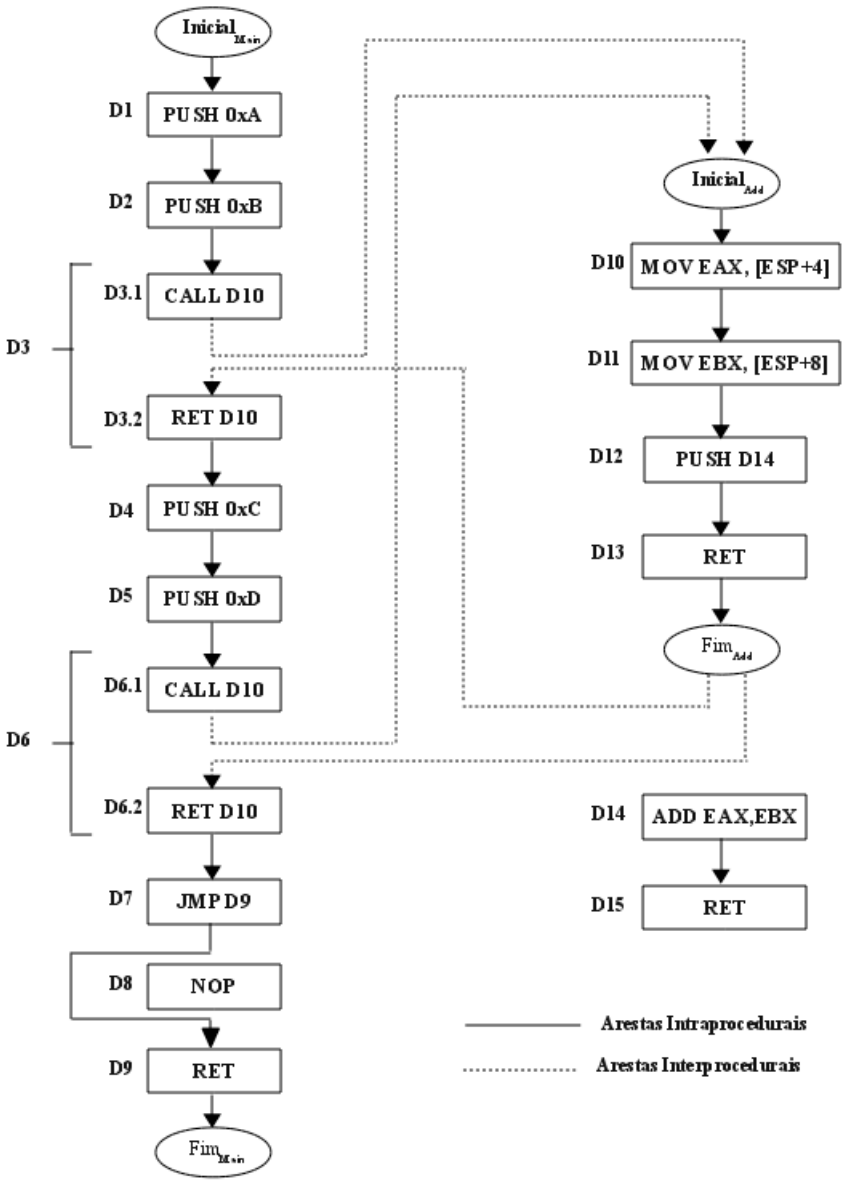

Figura 11. ICFG gerado após a aplicação da ofuscação de falso retorno

A Figura 11 mostra como o ICFG do programa ofuscado com falso retorno (Figura 10(b)) seria gerado.

Esse tipo de ofuscação corrompe a análise interprocedural de um programa porque induz a construção de arestas interprocedurais partindo de um nó final incorreto, criado a partir do falso retorno, para os nós de retorno de cada chamada de função. Além disso, o próprio CFG da função fica comprometido pois seu final não é identificado corretamente, ou seja, é identificado prematuramente. Nesta figura podemos ver que foram criadas arestas interprocedurais conectando o nó final de Add (Fim_{Add}) criado usando o falso retorno (D13) com os nós de retorno localizados em D3.2 e D6.2. Nesta figura também podemos ver como a análise intraprocedural foi comprometida, pois as instruções localizadas após D13 não fazem parte do *CFG* da função Add quando na verdade são instruções legítimas dessa função já que o fim da função Main está localizado na verdade em D15.

Ofuscação de Salto Incondicional

A ofuscação de salto incondicional é uma técnica de ofuscação que tem como objetivo substituir uma instrução de salto incondicional por uma chamada de função com a finalidade de dificultar a análise de um programa. Neste tipo de ofuscação a análise do programa torna-se mais difícil porque ao invés de um atacante descobrir de forma trivial para onde o fluxo de controle está sendo redirecionado, terá que analisar uma função, geralmente complexa, para obter a mesma informação. Tais funções são denominadas funções de redirecionamento[10].

A finalidade das funções de redirecionamento é redirecionar o fluxo de controle para o endereço correto, ou seja, o endereço para onde o fluxo de controle seria redirecionado caso a instrução de salto incondicional fosse mantida no programa. Entretanto a principal característica desse tipo de função é que elas são complexas de forma a dificultar ao máximo um atacante disposto a realizar a engenharia reversa, pois a análise dessa função obrigará ao atacante a gastar mais tempo e esforço para descobrir que essa função simplesmente redireciona o fluxo de controle para outro trecho do programa.

Na arquitetura x86, esse tipo de ofuscação pode ser realizado substituindo as instruções JMP por instruções CALL que invocam uma função de redirecionamento. No entanto, para garantir que tais funções redirecionem corretamente o fluxo de controle é necessário adicionar o destino da instrução de salto incondicional como parâmetro para essa função a fim de garantir que o fluxo de controle seja redirecionado corretamente.

A Figura 12 apresenta um exemplo que substitui a instrução JMP localizada em E7 do programa original (coluna à esquerda) por uma chamada de função como mostrado na coluna à direita em E8. Essa instrução realiza a chamada da função BF localizada em E14. Para garantir que o programa seja redirecionado corretamente pela função BF foi necessário informar o destino da instrução JMP que foi substituída. Com isso em mente foi adicionado a instrução "PUSH E9" (E7), ou seja, colocamos no topo da pilha o endereço da próxima instrução a ser executada RET (localizada em E9).

```
Main:                          Main:
      E1:   PUSH 0x1                 E1:   PUSH 0xA
      E2:   PUSH 0x2                 E2:   PUSH 0xB
      E3:   CALL Add                 E3:   CALL Add
      E4:   PUSH 0xC                 E4:   PUSH 0xC
      E5:   PUSH 0xD                 E5:   PUSH 0xD
      E6:   CALL Add                 E6:   CALL Add
      E7:   JMP B9                   E7:   PUSH E9
      E8:   NOP                      E8:   CALL BF
      E9:   RET                      E9:   RET
Add:                           Add:
      E10:  MOV eax,[esp+4]          E10:  MOV eax,[esp+4]
      E11:  MOV ebx,[esp+8]          E11:  MOV ebx,[esp+8]
      E12:  ADD eax,ebx              E12:  ADD eax,ebx
      E13:  RET                      E13:  RET
```

[10] Original do inglês, *branch function*.

```
                    BF:
                            E14:   MOV ecx,[esp+4]
                            E15:  JMP ecx
(a)                         (b)
```

Figura 12. Programa que exemplifica o uso da ofuscação de salto incondicional

Neste exemplo criamos uma função de redirecionamento que simplesmente redireciona o fluxo de controle para o endereço armazenado no topo da pilha. Em E14 a instrução "MOV ECX, [ESP+4]" copia o endereço armazenado no topo da pilha para o registrador ECX e em E15 o fluxo de controle é redirecionado para o endereço armazenado no registrador ECX, instrução "JMP ECX". No entanto para usufruir dessa técnica de ofuscação é necessário que a lógica que determina o endereço de retorno seja a mais complexa possível a fim de dificultar a análise por um atacante.

A Figura 13 mostra o *ICFG* criado a partir do programa ofuscado com a ofuscação de salto incondicional (Figura 12(b)). Essa figura mostra como o *ICFG* gerado é diferente do *ICFG* do programa original (Figura 5). No ICFG do programa ofuscado, foi criada uma aresta conectando a função Main e com a função BF, quando na verdade o fluxo de controle deveria ser redirecionado diretamente para um alvo específico, neste caso, para o endereço E9. Neste caso, as ferramentas de engenharia reversa deveriam esperar que após a execução de todas as instruções de BF, o fluxo de controle fosse redirecionado para o nó de retorno localizado em E8.2. No entanto isso não ocorre visto que não existe instrução de retorno da função BF, portanto não existe aresta interprocedural de retorno dessa função. Dessa forma, o atacante deverá entender a lógica dentro de BF a fim de identificar para onde o fluxo de controle está sendo redirecionado.

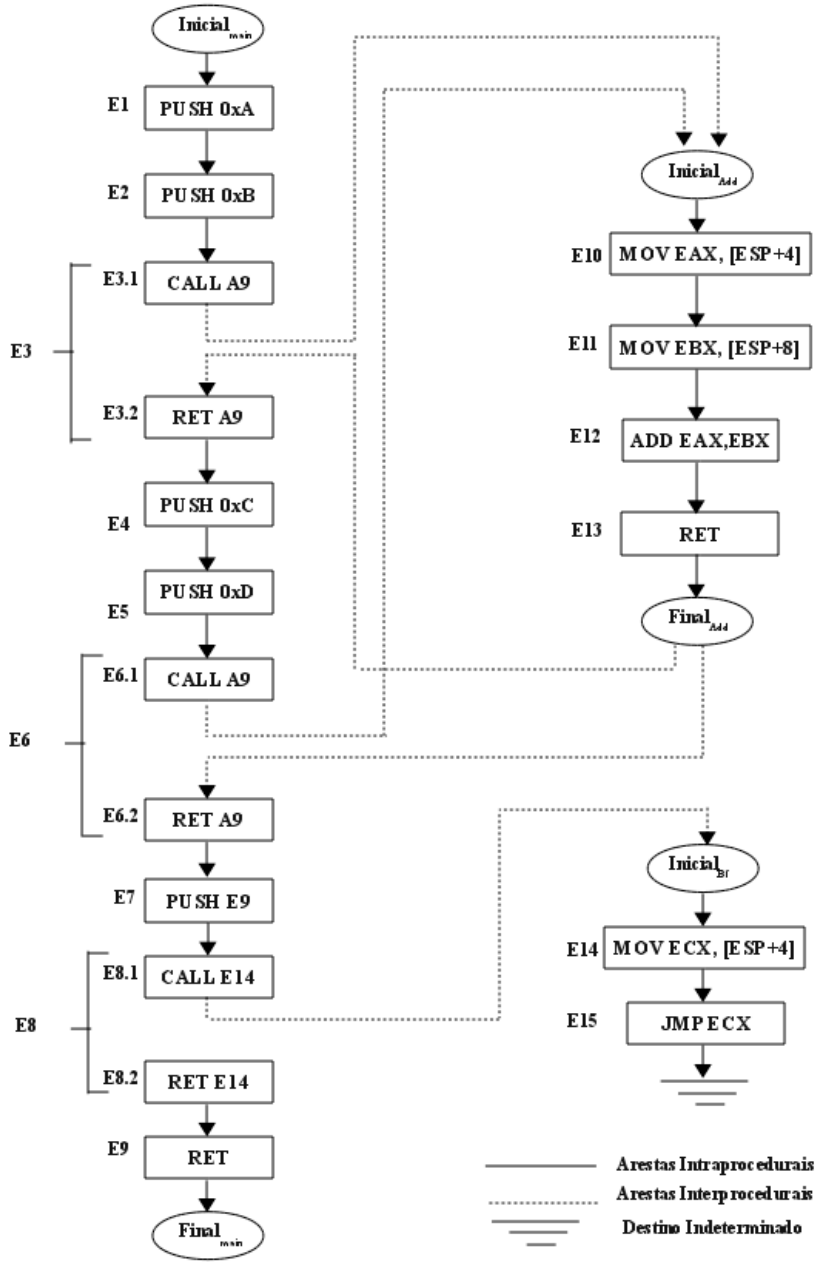

Figura 13. ICFG gerado após a aplicação da ofuscação de salto incondicional.

Predicado Opaco

Um predicado opaco é uma expressão que sempre retorna um valor conhecido independente do estado do programa ou das entradas fornecidas (COLLBERG e NAGRA 2009). Tal valor só é conhecido por quem está utilizando esse predicado. Um predicado opaco geralmente é utilizado para proteger um programa. Isso se deve porque ao inserir um predicado opaco antes de um salto condicional, um atacante deverá analisar ambos as ramificações desse salto condicional já que à princípio o atacante não sabe que o valor do predicado opaco será sempre o mesmo, forçando a execução de um dos ramos sempre. No entanto, isso faz com que o atacante perca tempo.

Um predicado opaco pode ser representado por uma expressão booleana, ou seja, uma expressão que retorna verdadeiro ou falso. No entanto diferentemente de outras expressões booleanas, o predicado opaco

sempre retorna um valor conhecido. A Figura 14 mostra uma representação gráfica de um predicado, onde P^V representa um predicado opaco que sempre retorna o valor verdadeiro (**Error! Reference source not found.** (a)) e P^F representa um predicado opaco que sempre retorna o valor falso (**Error! Reference source not found.** (b)). Note que as setas tracejadas na figura representam o caminho que nunca será alcançado durante a execução de um programa com esses predicados opacos.

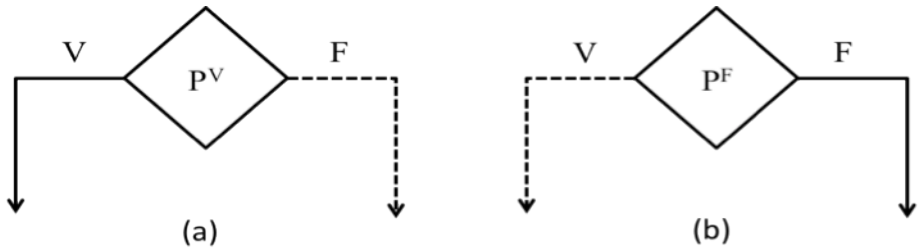

Figura 14. Predicado Opaco (a) Verdadeiro e (b) falso

1.5 Segurança em RSSFs

A segurança em RSSF*s* é um requisito tão importante quanto em qualquer outra rede de computadores. Contudo, devido as suas características e limitações e, as soluções de segurança para RSSF devem ser repensadas já que muitas das soluções de segurança utilizadas nas redes tradicionais não são aplicáveis para RSSF*s*.

Uma RSSF é um tipo de rede *ad hoc* utilizada para monitorar um ambiente, ou seja, coletar grandezas físicas tais como, temperatura, luminosidade e umidade com o intuito de fornecer essas informações para uma determinada aplicação que pode ser uma aplicação que detecta incêncios, diagnostica a saúde de estruturas ou controla o funcionamento de centrais de refrigeração. Os nós de uma RSSF*s* são dispositivos, denominados sensores, capazes de transformar propriedades físicas de um ambiente em sinais elétricos. Por exemplo, um nó sensor de uma RSSF é capaz de transformar o grau de luminosidade de um ambiente em sinais elétricos que em seguida são convertidos em sinais digitais para que possam ser manipulados por computadores. Os sensores são dispositivos que possuem recursos limitados, ou seja, pouco poder de processamento, pouca memória disponível e são alimentados por bateria. Por conta do baixo custo de produção desses sensores, as RSSF*s* podem ser compostas por um grande número de nós, frequentemente da ordem de centenas a milhares desses dispositivos que dispostos de forma distribuída fornecem informações para diversas aplicações.

A Figura 15 apresenta um modelo que representa os componentes de hardware de um típico nó sensor. Esse modelo é composto por quatro subsistemas: (i) subsistema de sensoriamento, (ii) subsistema de processamento, (iii) subsistema de comunicação e (iv) subsistema de energia (DELICATO 2005). O subsistema de sensoriamento é responsável por captar os dados analógicos do ambiente e convertê-los para sinais digitais. O subsistema responsável pelo processamento do sensor possui uma central de processamento e uma memória local que permitem a execução das tarefas. O subsistema de comunicação é encarregado da transmissão e recepção de informações entre os sensores. Atualmente a maioria dos sensores adota a tecnologia por rádio frequência para a comunicação. O subsistema de energia é basicamente composto por uma bateria, elemento essencial para uma RSSF já que define o tempo útil do sensor. Além dos componentes de hardware, um nó sensor possui um sistema operacional responsável pelo gerenciamento das operações do sensor. Um exemplo de sistema operacional desenvolvido especialmente para sensores, e utilizado em boa parte dos dispositivos existentes atualmente é o *TinyOS* (DELICATO 2005).

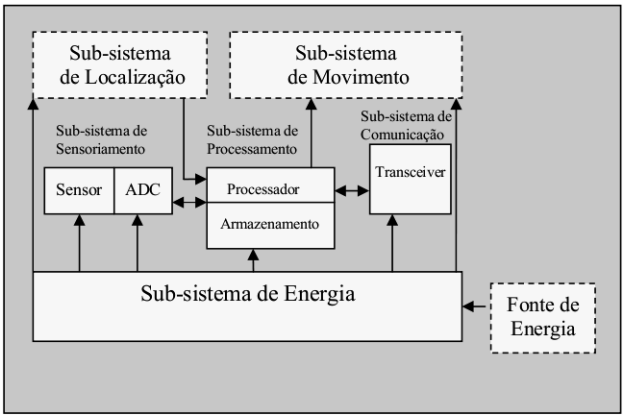

Figura 15. Componentes de Hardware de um nó sensor de uma RSSF

Uma RSSF, de forma geral, é constituída por apenas dois tipos de nós: nós comuns e nó sorvedouro, também chamado de estação base. Os nós comuns são responsáveis por captar os dados do ambiente e repassá-los para a estação base que por sua vez os armazena em um local seguro e os repassa para uma aplicação ou para outras redes.

As restrições de hardware e de energia dos sensores influenciam um projeto de uma aplicação para RSSF. Segundo Delicato (DELICATO 2005) um projeto de uma RSSF visa atender apenas os requisitos mínimos da aplicação, de forma a consumir menos energia e, assim, maximizar seu tempo de vida. Outra restrição em relação ao desenvolvimento de aplicações para RSSF*s* é em relação à limitação de recursos computacionais dos sensores, pois tais aplicações devem ser desenvolvidas considerando a disponibilidade de memória e de poder de processamento dos sensores. Outros aspectos que influenciam um projeto de aplicação para RSSF*s* são: tolerância a falhas, escalabilidade, topologia de rede, meio de transmissão e segurança.

Segundo Zia e Zomaya (ZIA e ZOMAYA 2006) a segurança de uma RSSF é dependente da aplicação e por isso os requisitos de segurança devem ser definidos a partir dos requisitos da aplicação e dos recursos dos sensores. As RSSF*s* estão sujeitas aos mesmos tipos de ataques de uma rede tradicional e de redes sem fio. Alguns desses ataques são descritos em seguida:

- **Coleta passiva de informações**[11]

Esse tipo de ataque é caracterizado pela captura de pacotes que estejam trafegando em um meio desprotegido;

- **Nó falso**[12]

Ataque caracterizado pela adição de nós na rede cuja finalidade é injetar informações falsas na rede;

- **Loops de roteamento**[13]

Esse tipo de ataque é caracterizado pela injeção ou reenvio de pacotes de roteamento maliciosos, mas válidos capazes de criar um loop de roteamento;

- **Repasse seletivo**[14]

[11] Original do inglês, *Passive Information Gathering*.
[12] Original do inglês, *False Node*.
[13] Original do inglês, *Routing Loops*.

Ataque onde um nó malicioso realiza o descarte de determinados pacotes com a finalidade de aumentar a latência da rede;

- **Ataque de negação de serviço**[15]

Ataque caracterizado pelo congestionamento no enlace de comunicação, capaz de impedir a comunicação entre os sensores;

- **Ataque de *Sybil***[16]

Ataque caracterizado pela presença de múltiplas identidades (roubadas ou fabricadas) em um único nó malicioso;

- **Comprometimento de nó**[17]

Ataque caracterizado por um atacante que ao obter acesso físico a um nó pode ser capaz de extrair dados confidenciais como chaves criptográficas ou ainda adulterar o funcionamento da RSSFs ao substituir nós dessa rede por nós comprometidos e controlados pelo atacante.

O arcabouço de ofuscação proposto nesse trabalho tem como objetivo combater esse tipo de ataque a fim de proteger os sensores contra atacantes dispostos a realizar a engenharia reversa dos programas em execução nesses sensores.

Trabalhos Relacionados

Nesta seção são apresentados trabalhos da literatura relacionados com algoritmos de desmontagem, ofuscação de código e ofuscação de código aplicada para proteger RSSFs.

1.6 Algoritmos de Desmontagem

A fase de desmontagem como já mencionado anteriormente é a fase inicial da engenharia reversa. Portanto qualquer comprometimento nessa fase pode acarretar em erros para as demais fases da engenharia reversa, pois

---

[14] Original do inglês, *Selective Forward*.
[15] Original do inglês, *Denial of Service Attack* ou *DoS*
[16] Original do inglês, *Sybil Attack*
[17] Original do inglês, *Node Compromise*

estas utilizam o código *assembly* gerado pela desmontagem. Os dois principais algoritmos utilizados para desmontagem são: (i) varredura linear e (ii) transversal recursivo (SCHWARZ *et al*. 2002).

O algoritmo varredura linear (SCHWARZ *et al*. 2002) tenta traduzir os bytes iniciados em um endereço, para isso verifica se esses bytes correspondem a um código de uma instrução válida para a arquitetura de hardware que esse programa foi compilado. Esse processo de tradução segue linearmente até que o ultimo endereço do programa seja atingido. O principal problema do algoritmo varredura linear é que ele é suscetível a erros durante a desmontagem de um programa caso sejam adicionados dados no segmento de código do programa já que se espera nesse segmento somente instruções. Nesse caso, os dados serão traduzidos como se fossem instruções. A partir desse pressuposto, propomos a inserção de dados confidenciais no segmento de código do programa com o intuito de camuflar esses dados, ou seja, induzir os as ferramentas de engenharia reversa a traduzirem esses dados como se fossem instruções.

O algoritmo transversal recursivo (SCHWARZ *et al*. 2002) assim como o algoritmo varredura linear inicia a tradução dos bytes a partir do primeiro endereço do programa. No entanto a tradução não segue linearmente, pois toda vez que for identificada uma instrução que realiza um redirecionamento no fluxo de controle, o algoritmo tenta identificar o alvo dessa instrução e tenta continuar a tradução a partir desse endereço. Apesar de considerar o fluxo de controle ao realizar a desmontagem, esse algoritmo ainda é suscetível a erros de desmontagem, por exemplo, caso haja imprecisões ao estimar o endereço da próxima instrução a ser traduzida, a desmontagem podem ser realizada incorretamente. Desse modo, nesse trabalho, utilizamos técnicas de ofuscação capazes de ofuscar o fluxo de controle de um programa comprometendo tanto o algoritmo varredura linear quanto o algoritmo transversal recursivo.

1.7 Ofuscação de Código

A maioria dos trabalhos sobre ofuscação concentra seus esforços em técnicas capazes de comprometer a fase de descompilação da engenharia reversa (WANG 2000), (CHO *et al*. 2001) e (OGISO *et al*. 2003). Tais trabalhos buscam basicamente dificultar a análise de um programa através de mecanismos capazes de dificultar a construção do *CFG* e do *CG* de um programa. Entretanto o presente trabalho utiliza técnicas de ofuscação capazes de comprometer a fase de desmontagem de um programa, sendo, portanto complementar a essas propostas. Nos trabalhos de Linn e Debray (LINN e DEBRAY 2003), Popov *et al*. (POPOV *et al*. 2007) e Lee *et al*. (LEE *et al*. 2010) são propostas técnicas de ofuscação capazes de comprometer tanto o algoritmo varredura linear quanto o algoritmo transversal recursivo.

Linn e Debray (LINN e DEBRAY 2003) aproveitaram do fato de que as ferramentas de engenharia reversa assumem certas convenções de compilação durante a desmontagem de um programa. Por exemplo, elas assumem que o endereço de retorno de uma função é o endereço imediatamente depois do endereço da instrução que realizou a chamada dessa função. A partir disso, Linn e Debray propuseram a inserção de código morto[18], bytes que não representam instruções nem dados do programa, após uma chamada de função cujo endereço de retorno foi manipulado, comprometendo a desmontagem desse programa, já que os algoritmos de desmontagem tentarão traduzir o código morto como se fosse instrução. Outra técnica proposta por Linn e Debray é denominada ofuscação de salto incondicional que substitui as instruções de salto incondicionais por chamadas de

[18] Original do inglês, *junky bytes*

função para um determinado tipo de função (função de redirecionamento). Esse tipo de função compromete a desmontagem de um programa porque as funções de redirecionamento não se comportam de forma esperada, ou seja, não obedece as convenções de compilação quando ocorre uma chamada e retorno de função. Muitas das técnicas de ofuscação utilizadas no presente trabalho são versões personalizadas das técnicas propostas por Linn e Debray.

Popov *et al*. (POPOV *et al*. 2007), por sua vez, propuseram um mecanismo capaz de ofuscar o fluxo de controle de um programa combinando o uso de sinais de interrupção e o tratamento de exceções. Um sinal de interrupção é uma forma de comunicação entre processos de um sistema operacional e o tratamento de exceções é o elemento responsável por decidir o que fazer quando ocorrem tais interrupções, responsáveis por alterar o fluxo normal de execução do programa. O mecanismo proposto por Popov *et al*. substitui instruções que mudam o fluxo de controle, tais como *JMP*, *CALL* e *RET*, por instruções capazes de causar esses sinais a fim de ativar um tratamento de exceções personalizado que irá invocar uma função de restauração, responsável por transferir o fluxo de controle para o alvo correto, ou seja, para o alvo da instrução que foi substituída (endereço destino da instrução de salto incondicional, por exemplo). Uma das vantagens desse mecanismo é que ele é capaz de criar áreas do programa que nunca serão executadas. Tais áreas estão localizadas logo após as instruções que causam os sinais e podem ser utilizadas para adicionar código morto ou outras instruções espúrias cuja finalidade é somente dificultar a análise do programa. No presente trabalho também foram criadas áreas no segmento do código do programa que nunca serão executadas. Contudo, no presente trabalho utilizamos essas áreas para armazenar/esconder dados confidenciais do programa.

Lee *et al*. (LEE *et al*. 2010) propuseram um arcabouço capaz de transformar um programa em duas fases. Na primeira fase, o arcabouço aplica técnicas de ofuscação capazes de comprometer a desmontagem do programa e na segunda fase, utiliza um método para induzir que um programa ofuscado se pareça com um programa não ofuscado. A ofuscação proposta por Lee *et al*. é realizada adicionando instruções capazes de lançar exceções em qualquer parte do programa. Tais exceções são tratadas por um tratador de exceções personalizado que é responsável por redirecionar o fluxo de controle do programa, assim como na proposta de Popov *et al*. Na segunda fase, o tratador de exceções que também capaz de criar áreas do programa que nunca serão executadas, permite com que instruções sejam adicionadas com o intuito de balancear a distribuição de instruções a fim de manter essa distribuição semelhante a distribuição de um programa não ofuscado. No presente trabalho, também foi proposto um arcabouço de ofuscação. No entanto a arquitetura do nosso arcabouço pode ser personalizada para qualquer arquitetura de hardware. Diferentemente do trabalho de Lee *et al*., no presente trabalho não precisamos utilizar um método para fazer com que um programa ofuscado se pareça com um programa não ofuscado já que as técnicas de ofuscação utilizadas já são furtivas por natureza. Outro ponto que diferencia nosso trabalho do trabalho de Lee *et al*. é que ele as áreas do segmento de código que nunca são executadas são criadas combinando as técnicas de ofuscação com a manipulação no fluxo de controle.

1.8 Ofuscação aplicada em RSSFs

Existem muitos trabalhos na literatura que propõem soluções para melhorar a segurança de RSSFs. Tais como (PARK e SHIN 2005), (PELISSIER *et al*. 2011), (ALARIFI e DU 2006) e (GU 2010). Entretanto somente alguns desses trabalhos como os de Alarifi e Du (ALARIFI e DU 2006) e Gu (GU 2010) apresentam propostas para combater ataques que buscam comprometer um nó sensor utilizando ofuscação de código.

Alarifi e Du (ALARIFI e DU 2006) apresentaram um método de proteção para os nós sensores que combina ofuscação com aleatorização de partes do código mantendo sua funcionalidade original. Neste trabalho, a ofuscação é aplicada em dois momentos distintos. No primeiro momento, dados confidenciais são ofuscados utilizando determinadas funções que transformam o código e, em um segundo momento, tais funções também são ofuscadas visto que essas funções também devem ser armazenadas dentro do programa, pois são utilizadas para acessar os dados confidenciais do programa. A segurança dos dados confidenciais proposto por Alarifi e Du se baseia na segurança das funções utilizadas para ofuscar esses dados e caso essas funções sejam descobertas, os dados confidenciais poderão ser revelados. Entretanto, o presente trabalho propõe o uso de técnicas de ofuscação mais resistente à desofuscadores, ou seja, técnicas baseadas na substituição e inserção de instruções de forma aleatória, que misturadas com as demais instruções do programa, dificultam o desenvolvimento de desofuscadores automáticos capazes de identificar os pontos do programa onde foram ofuscados. Diferentemente da nossa proposta, Alarifi e Du aplicam ofuscações que comprometem a fase de descompilação. Portanto as técnicas de ofuscação utilizadas no presente trabalho, ofuscações que comprometem a fase de desmontagem, atuam de forma complementar a proposta de Alarifi e Du.

A abordagem utilizada por Gu para melhorar a segurança das RSSFs baseia-se em diversificar os programas implantados nos nós de uma RSSF. A ideia é aplicar técnicas de ofuscação que modificam o código executável dos sensores sem modificar o seu funcionamento a fim de criar versões distintas de um mesmo programa, um para cada sensor. Essa abordagem visa dificultar o trabalho de um atacante que esteja disposto a extrair dados ou mesmo modificar o funcionamento de vários sensores porque todo o esforço utilizado para analisar o código de um sensor não será aproveitado para comprometer outros sensores dessa mesma rede. Diferente do presente trabalho, as técnicas de ofuscação utilizadas por Gu buscam comprometer a fase de descompilação. Nossa proposta, por utilizar técnicas capazes de comprometer a fase de desmontagem complementa o trabalho de Gu, pois qualquer comprometimento na fase de desmontagem poderá tornar o resultado da descompilação ainda mais incorreto já que a descompilação depende do código *assembly* gerado pela desmontagem.

Arcabouço de Ofuscação de Código e Proteção de Dados

Nesta seção apresentamos a descrição de um arcabouço de ofuscação de código e proteção de dados útil para combater ataques de comprometimento de nó. Esse arcabouço é ***eficaz*** porque as técnicas de ofuscação empregadas por ele dificultam a extração de dados e adulteração de código. É ***furtivo***, pois um programa ofuscado com esse arcabouço não dá indício de que está ofuscado. É ***eficiente*** porque as suas ofuscações não impactam de forma negativa nos recursos computacionais, ou seja, não aumenta muito o tamanho do programa e nem o número de ciclos de processamento, pois essas ofuscações são baseadas na substituição e inserção de instruções. Por ultimo esse arcabouço pode ser considerado ***resistente à desofuscadores*** porque um programa ofuscado com ele é capaz de resistir a ataques realizados por desofuscadores automáticos.

A arquitetura desse arcabouço pode ser personalizada para qualquer arquitetura de hardware desde que as técnicas de ofuscação sejam personalizadas adequadamente, ou seja, exista as informações necessárias para realizar as substituições capazes aplicar tais técnicas de ofuscação para a arquitetura de hardware em questão. Outra vantagem desse arcabouço é que essas ofuscações podem ser aplicadas em qualquer programa, pois as substituições realizadas mantém a mesma semântica do programa.

Tanto as técnicas de ofuscação quanto o mecanismo de proteção de dados desse arcabouço são utilizados em tempo de projeto, ou seja, antes que o programa ofuscado seja implantado nos dispositivos. Dessa forma esse arcabouço não interfere no funcionamento da rede onde esses dispositivos estão conectados.

Nas próximas seções, serão descritos: (i) a arquitetura lógica do arcabouço, apresentando seus componentes de software e suas principais características; (ii) uma descrição detalhada dos componentes desse arcabouço; e (iii) a operação desse arcabouço mostrando como os componentes dessa arquitetura se relacionam.

1.9 Arquitetura Lógica

A arquitetura lógica representada através do diagrama de componentes, mostrado na Figura 16 consiste de cinco componentes: Gestor de Ofuscação, Gestor de Segurança, Compilador, Ofuscador e Protetor de Dados; e três bases de dados: Regras de Segurança, Perfis de Segurança e Regras de Ofuscação.

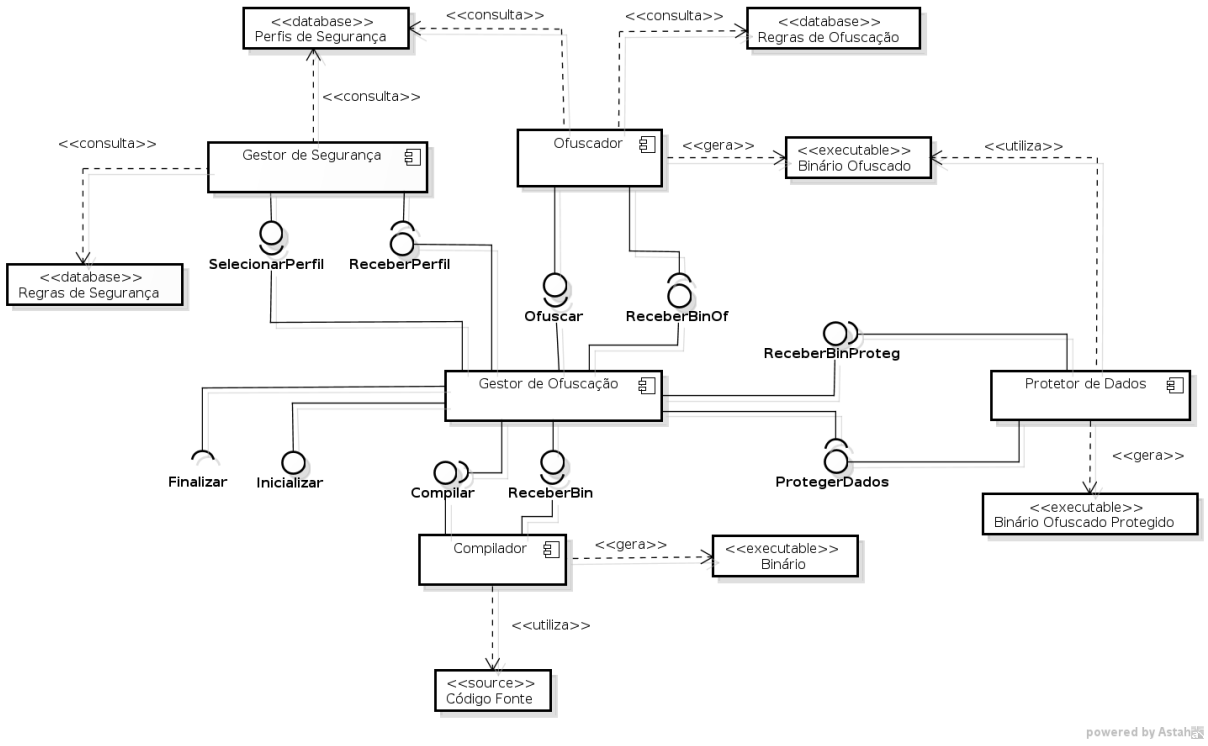

Figura 16. Diagrama de componentes da arquitetura lógico do arcabouço de ofuscação de código e proteção de dados

O funcionamento desse arcabouço depende das seguintes informações providas pelo usuário: **Código Fonte** (endereço onde está localizado o código fonte do programa que será ofuscado), **Lista de Dados Confidenciais** (estrutura de dados que identifica os dados do programa que precisam ser protegidos), **Plataforma de Hardware** (estrutura de dados que define a arquitetura de hardware utilizada nos dispositivos onde o programa ofuscado irá executar) e o **Nível de Segurança** (valor que define o nível de segurança que deve ser aplicada com o intuito de garantir o mínimo de segurança ao programa). Por exemplo, o Nível de Segurança pode ser definido em três níveis: mínimo, intermediário e máximo. Portanto caso seja escolhido o Nível de Segurança máximo, serão aplicadas em um programa todas as ofuscações possíveis. É bom ressaltar que o Nível de Segurança deve ser fornecido por um especialista, ou seja, uma pessoa que conheça os requisitos de segurança do programa a ser ofuscado. Durante o funcionamento desse arcabouço os seguintes arquivos serão gerados:

Binário (Código Fonte compilado), **Binário Ofuscado** (Binário transformado através das técnicas de ofuscação) e **Binário Ofuscado Protegido** (Binário Ofuscado onde os dados sensíveis do programa foram protegidos).

O **Gestor de Ofuscação** é o componente responsável por gerenciar toda ação dos outros componentes dessa arquitetura. O componente **Gestor de Segurança** é o elemento responsável por selecionar o Perfil de Segurança adequado para um determinado Código Fonte de acordo com o Nível de Segurança escolhido. O **Perfil de Segurança** define quais **Técnicas de Ofuscação** (técnicas que substituem ou adicionam instruções como a ofuscação de chamada, ofuscação de retorno, ofuscação de falso retorno e ofuscação de salto incondicional) serão utilizadas e o **Grau de Ofuscação** (número de vezes em que uma determinada Técnica de Ofuscação é aplicada em um programa). O **Compilador** é o componente responsável pela manipulação e compilação do Código Fonte. O **Ofuscador**, por sua vez, é o componente responsável por transformar o Binário. Esse componente o transforma de acordo com as Técnicas de Ofuscação definidas pelo Perfil de Segurança. Outra tarefa do Ofuscador é armazenar na Lista de Trechos não Executáveis de Código as informações referentes aos endereços onde os dados confidenciais poderão ser armazenados. Por ultimo O **Protetor de Dados** é o componente responsável por armazenar de forma segura os dados confidenciais do programa.

A base de dados, denominada **Perfis de Segurança**, especifica os Perfis de Segurança adequados a fim de atingir os Níveis de Segurança. A base Perfis de Segurança contém os seguintes campos: identificação do Perfil de Segurança, conjunto de Técnicas de Ofuscação e o Grau de Ofuscação a serem aplicados em um programa. Cada Perfil de Segurança identifica a combinação das Técnicas de Ofuscação com um determinado Grau de Ofuscação.

A base de dados, denominada **Regras de Segurança**, contém as regras utilizadas pelo componente Gestor de Segurança para mapear qual Perfil de Segurança deve ser escolhido dado um determinado Nível de Segurança necessário. Os campos dessa base de dados são: identificação da regra, identificação do Nível de Segurança e identificação do Perfil de Segurança.

A base de dados, denominada **Regras de Ofuscação**, é responsável por armazenar as informações utilizadas pelo componente Ofuscador para realizar as transformações necessárias a fim de ofuscar um Binário que foi compilado para uma determinada Plataforma de Hardware. Os campos dessa base de dados são: identificação da Técnica de Ofuscação, conjunto de instruções que serão substituídas e o conjunto de novas instruções que substituirão o conjunto de instruções anterior. Essa base de dados deve ser preenchida por um especialista de segurança que conheça a Plataforma de Hardware para o qual essa arquitetura será personalizada. Tal base de dados é única para cada Plataforma de Hardware e a tarefa do especialista de segurança personalizar cada uma das Técnicas de Ofuscação para o conjunto de instruções da arquitetura utilizada pela respectiva Plataforma de Hardware. É bom ressaltar que todas as bases de dados desse arcabouço devem ser previamente preenchidas durante a fase de *set-up*, ou seja, fase em que o arcabouço está sendo instanciado.

Durante a execução do arcabouço algumas estruturas de dados são criadas. Tais estruturas de dados são denominadas: Lista de Dados Confidenciais e Lista de Trechos não Executáveis de Código. Cada elemento da **Lista de Dados Confidenciais** possui duas informações: identificador e o tamanho do dado confidencial. Tais informações permitem ao Protetor de Dados identificar quais dados do programa são confidenciais e devem ser protegidos. A **Lista de Trechos Não Executáveis de Código**, por sua vez, é uma estrutura de dados que

armazena os endereços do Binário Ofuscado onde os dados confidenciais podem ser armazenados de forma segura. Cada elemento dessa estrutura de dados possui duas informações: endereço do Binário Ofuscado e sua capacidade (informa qual é o tamanho máximo que um dado deve ter para ser armazenado a partir do endereço do Binário Ofuscado fornecido).

Descrição do Gestor de Ofuscação

O Gestor de Ofuscação é o componente responsável por gerenciar toda ação dos outros componentes do *TinyObf*. Este componente possui as seguintes interfaces: Inicializar, ReceberPerfil, ReceberBin, ReceberBinOf, ReceberBinProteg e Finalizar. A interface **Inicializar** permite com que um usuário solicite o início desse arcabouço. Essa interface requer as seguintes informações como parâmetros: Código Fonte, Lista de Dados Confidenciais e o Nível de Segurança a fim de ofuscar o código e proteger os dados confidenciais de um programa. A interface **ReceberPerfil** é responsável por receber o Perfil de Segurança do Gestor de Segurança. Essa interface possui um único parâmetro: Perfil de Segurança. A interface **ReceberBin** é responsável por receber o Binário do Compilador. Portanto seu único parâmetro é a localização desse Binário. A interface **ReceberBinOf**, por sua vez, possui os seguintes parâmetros: Binário Ofuscado e a Lista de Trechos não Executáveis de Código. Essa interface é responsável por receber a localização do Binário Ofuscado e os endereços dos trechos não executáveis de código do Ofuscador. A interface **ReceberBinProteg** é responsável por receber a localização do Binário Ofuscado Protegido do Protetor de Dados, sendo esse o seu único parâmetro. Por ultimo a interface **Finalizar** é responsável por informar ao usuário que a ofuscação de código e a proteção de dados foram concluídas. Essa interface entrega o Binário Ofuscado Protegido que poderá ser implantado nos dispositivos desejados.

Descrição do Gestor de Segurança

O Gestor de Segurança é o componente que é responsável por escolher qual o Perfil de Segurança é capaz de garantir um determinado Nível de Segurança (informação fornecida pelo usuário). O Gestor de Segurança provê a interface **SolicitarPerfil** que deve receber como parâmetro o Nível de Segurança. Com essa informação, o Gestor de Segurança consulta a base de dados Regras de Segurança a fim de identificar qual Perfil de Segurança corresponde ao Nível de Segurança fornecido como entrada.

Após selecionar o Perfil de Segurança adequado, o Gestor de Segurança o fornece para o Gestor de Ofuscação através da interface ReceberPerfil.

Descrição do Compilador

O Compilador é o componente responsável pela manipulação e compilação do Código Fonte. O Compilador deve manipular o Código Fonte a fim de tornar o Binário apto para ser ofuscado. Essa manipulação é realizada por causa da dificuldade em inserir instruções em um código executável, pois inserir instruções no meio do código de um programa pode alterar o destino das instruções de redirecionamento. Por exemplo, ao inserir uma instrução entre uma instrução de redirecionamento e o seu alvo, a instrução de redirecionamento não será capaz de redirecionar corretamente o fluxo de controle porque o endereço do alvo será modificado. Outro exemplo seria a inserção de uma instrução antes de uma instrução de redirecionamento e seu alvo. Nesse segundo exemplo, a instrução de redirecionamento também não será capaz de redirecionar corretamente o fluxo de controle, pois o endereço de ambos será modificado. É bom ressaltar que a remoção de instruções também

podem causar os mesmos problemas que a inserção de instruções.

A reescrita do código executável, ou seja, a remoção ou inserção de instruções apesar de parecer uma tarefa impossível pode ser feita através de ajustes no código utilizando as informações de realocação de um programa (caso essas informações estejam disponíveis). No entanto existem trabalhos na literatura que realizam a reescrita do código executável sem utilizar essas informações como (SMITHSON *et al*. 2010) e (O'SULLIVAN *et al*. 2011). O'Sullivan *et al*. (O'SULLIVAN *et al*. 2011) propõe a reescrita de um código executável a fim de adicionar medidas de segurança em um programa. No presente trabalho, optou-se em manipular diretamente o Código Fonte a fim de permitir a inserção de instruções quando for necessário aplicar as técnicas de ofuscação. Tal escolha foi feita porque essa é a abordagem mais simples já que simplesmente adicionamos instruções NOP (No Operation – instruções que não faz nada) suficientes após as instruções que serão substituídas. O número de instruções NOP pode variar dependendo da técnica de ofuscação que será aplicada, pois cada técnica de ofuscação pode substituir uma instrução por conjuntos de instruções de tamanho diferente.

Após a manipulação do Código Fonte, o Compilador ajusta suas configurações a fim de gerar o Binário adequado para o dispositivo onde esse código executável será implantado. Basicamente esses ajustes são realizados a fim de selecionar a Plataforma de Hardware. O Compilador pode ser capaz de compilar um Código Fonte para diferentes Plataformas de Hardware desde que os demais componentes sejam capazes de tratar o Binário gerado.

O Compilador provê a interface Compilar cujos parâmetros são: Código Fonte e Plataforma de Hardware. Através dessa interface, o Gestor de Ofuscação solicita ao Compilador que um Código Fonte seja compilado para uma determinada Plataforma de Hardware. Após gerar o Binário, o Compilador utiliza a interface ReceberBin do Gestor de Ofuscação para informá-lo da localização do Binário Ofuscado.

Descrição do Ofuscador

O Ofuscador é o componente responsável por realizar todas as transformações necessárias no Binário a fim de ofuscá-lo. Após a execução desse componente, é gerado um código executável, denominado Binário Ofuscado, que possui características capazes de comprometer as ferramentas de engenharia reversa.

As técnicas de ofuscação empregadas por esse arcabouço comprometem as ferramentas de engenharia reversa porque exploram indevidamente as convenções de compilação, acarretando erros tanto na desmontagem do código executável quanto na geração de estruturas (ICFG, CG e CFGs) utilizadas nas análises intra e interprocedurais. Dessa forma a análise do programa torna-se suscetível a erros já que o código desmontado e as estruturas utilizadas nas análises desse programa não são confiáveis. Portanto todo tempo e esforço do atacante para extrair informações poderá ser desperdiçado.

As transformações no Binário são realizadas de acordo com o Perfil de Segurança. Portanto antes de realizar as transformações, o Ofuscador consulta a base de dados Perfis de Segurança a fim de identificar quais Técnicas de Ofuscação e Grau de Ofuscação devem ser utilizados. Após esse passo, o Ofuscador consulta a base de dados Regras de Ofuscação para identificar para cada Técnica de Ofuscação, quais instruções deverão ser substituídas e quais serão as novas instruções que serão utilizadas para ofuscar o Binário.

As Técnicas de Ofuscação utilizadas nesse arcabouço podem ser combinadas com a manipulação do

fluxo de controle a fim de criar trechos no segmento de código de um programa que nunca são executadas, denominados trechos de código não executáveis[19]. Esses trechos podem ser utilizados para armazenar dados confidenciais como mostrado em (COSTA *et al.* 2012), pois os bytes localizados nesses trechos são traduzidos como se fossem instruções válidas pelas ferramentas de engenharia reversa. Assim, os trechos de código não executáveis podem dificultar a descoberta de dados confidenciais já que os atacantes deverão descobrir dentre as instruções do programa, quais são na verdade dados confidenciais que estão camuflados como se fossem instruções.

```
Main:                              Main:
      L1:   PUSH 0xA                     M1:   PUSH 0xA
      L2:   PUSH 0xB                     M2:   PUSH 0xB
      L3:   CALL Add                     M3:   PUSH M8
      L4:   NOP                          M4:   PUSH M9
      L5:   NOP                          M5:   RET
      L6:   NOP                          M6:   NOP   } Trecho de Código não Executável
      L7:   NOP                          M7:   NOP   }
      L8:   RET                          M8:   RET
Add:                               Add:
      L9:   MOV eax,[esp+4]              M9:   MOV eax,[esp+4]
      L10:  MOV ebx,[esp+8]              M10:  MOV ebx,[esp+8]
      L11:  ADD eax,ebx                  M11:  ADD eax,ebx
      L12:  RET                          M12:  RET
            (a)                                (b)
```

Figura 17. Criação de trechos não executáveis de código. (a) programa original e (b) programa ofuscado com ofuscação de chamada e onde foi criado um trecho de código não executável

A Figura 17 mostra como criar um trecho de código não executável utilizando a ofuscação de chamada. Na coluna à esquerda (Figura 17(a)) é mostrado um programa onde será criado o trecho de código não executável. Na coluna à direita (Figura 17(b)) foi aplicada a ofuscação de chamada na instrução de chamada localizada em L3 e, em seguida, combinou-se com a manipulação do fluxo de controle a fim de criar um trecho de código não executável entre os endereços M6 e M7. A instrução PUSH (M3) empilha o endereço de retorno M8 e a segunda instrução PUSH (M4) empilha o endereço da função a ser executada, ou seja, M9. Quando a instrução RET no endereço M5 executa, o fluxo de controle é redirecionado para o endereço inicial da função Add (M9). Após executar as instruções da função Add, o fluxo de controle é redirecionado para o endereço localizado no topo da pilha, neste caso, M8. Com isso os endereços M6 e M7 nunca são executados e com isso podem ser considerados como um trecho de código não executável, pois todos os bytes armazenados nesses endereços serão traduzidos como se fossem instruções.

O Ofuscador, antes de realizar as ofuscações, percorre o código do Binário tentando identificar os endereços das instruções que devem ser substituídas. Em seguida, ele consulta a Lista de Dados Confidenciais para descobrir o tamanho dos dados confidenciais que deverão ser protegidos. Através do tamanho desses dados, o Ofuscador é capaz de manipular corretamente o fluxo de controle de forma a criar os trechos de código não executáveis capazes de armazenar os dados confidenciais do programa de forma segura, ou seja, camuflado em

[19] Original do inglês, *dead execution spots*

meio as demais instruções do programa. Por último, o Ofuscador realiza as transformações no Binário gerando o Binário Ofuscado e armazena os endereços dos trechos não executáveis de código em uma estrutura de dados denominada Lista de Trechos não Executáveis de Código.

O Ofuscador provê a interface Ofuscar que possui os seguintes parâmetros: a localização do Binário, o Perfil de Segurança e a Lista de Dados Confidenciais. Através dessa interface, o Gestor de Ofuscação solicita ao Ofuscador que um Binário seja ofuscado e que sejam criados trechos de código não executáveis capazes de armazenar os dados listados na Lista de Dados Confidenciais. Após a geração do Binário Ofuscado e da Lista de Trechos não Executáveis de Código, o Ofuscador envia essas informações ao Gestor de Ofuscação através da interface ReceberBinOf.

Descrição do Protetor de Dados

A tarefa do Protetor de Dados é armazenar de forma segura os dados contidos na Lista de Dados Confidenciais. Tais dados devem ser armazenados nos trechos de código não executáveis criados pelo Ofuscador a fim de que eles sejam traduzidos como se fossem instruções, dessa forma passem despercebidos pelos atacantes dispostos a realizar a engenharia reversa.

Os dados confidenciais armazenados em trechos de código não executáveis devem ser traduzidos como se fossem instruções válidas. Para tal, antes de inserir os dados confidenciais, o Protetor de Dados se certifica que esses bytes correspondem a um código de uma instrução. Caso isso não seja verdade, o Protetor de Dados insere bytes que identificam o operador de um *opcode* válido de maneira que os bytes do dado confidencial passem a ser os operandos desse *opcode*. Caso o tamanho desses dados confidenciais seja maior que o tamanho em bytes dos operandos desse *opcode*, é acrescentado um novo byte que identifica outro operador e o procedimento é descrito anteriormente é repetido. Com isso, os dados confidenciais sempre serão camuflados em meio as demais instruções do programa.

O primeiro passo do Protetor de Dados para proteger os dados confidenciais é verificar quais endereços do Binário Ofuscado, especificado pela Lista de Trechos não Executáveis de Código, podem armazenar os dados contidos na Lista de Dados Confidenciais. Após esse passo, o Protetor de Dados armazena os dados confidenciais nesses trechos de maneira que esses bytes sejam sempre traduzidos como instruções válidas. Em seguida o Protetor de Dados modifica todas as referências aos dados confidenciais, antes localizados no segmento de dados do programa, a fim de manter o mesmo funcionamento do programa, pois esses dados foram realocados dentro do código executável do programa. Ao final, o Protetor de Dados gera o arquivo Binário Ofuscado Protegido (arquivo que será retornado ao usuário e que poderá ser implantado nos dispositivos).

O Protetor de Dados provê a interface ProtegerDados. Essa interface possui os seguintes parâmetros: endereço do Binário Ofuscado, Lista de Dados Confidenciais e Lista de Trechos não Executáveis de Código. Essa interface permite com que o Gestor de Ofuscação solicite quais dados confidenciais devem ser protegidos. Após armazenar os dados confidenciais de forma segura, o Protetor de dados utiliza a interface ReceberBinProteg para informar ao Gestor de Ofuscação a localização do Binário Ofuscado Protegido.

1.10 Operação

Toda a operação realizada pelos componentes desse arcabouço ocorre em tempo de projeto, ou seja, antes do programa ser implantado nos dispositivos e com isso antes do programa estar em execução. A **Error! Reference**

source not found.

apresenta o diagrama de sequência que representa toda a operação desse arcabouço de ofuscação desde a sua primeira interação com o usuário até a entrega do Binário Ofuscado Protegido.

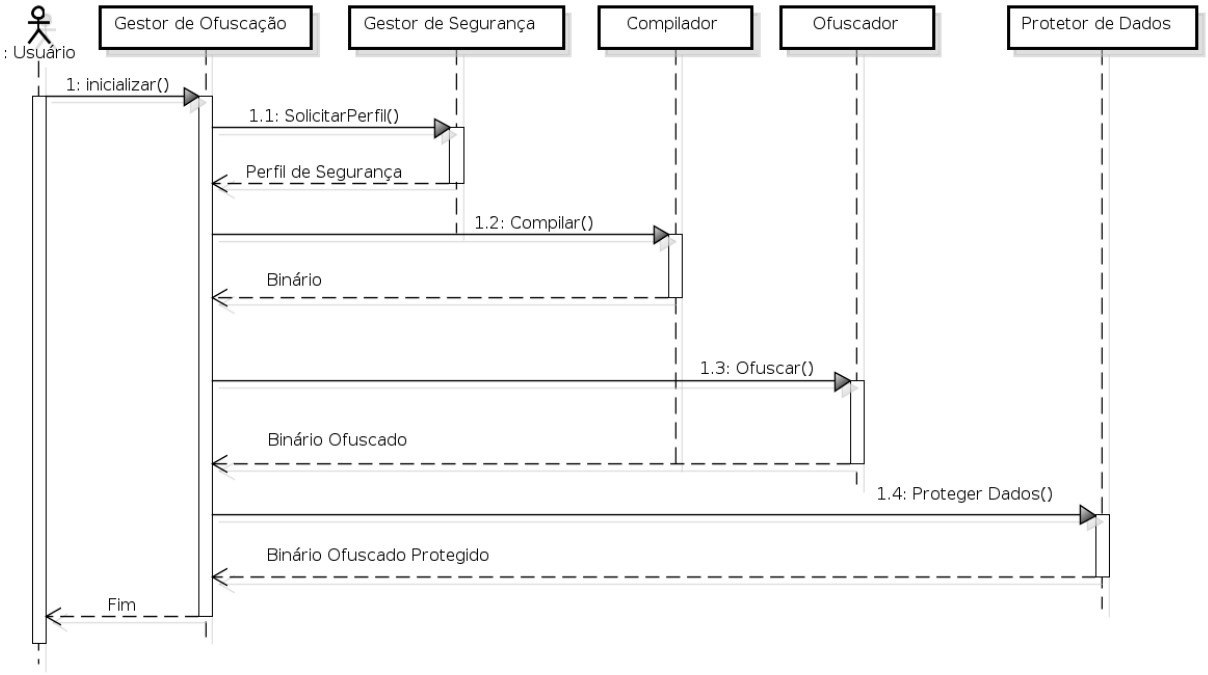

Figura 18. Diagrama de sequência do arcabouço de ofuscação

O arcabouço inicia sua execução recebendo como entrada as seguintes informações: Código Fonte, Nível de Segurança, Plataforma de Hardware e Lista de Dados Confidenciais. Tais informações devem ser encaminhadas para o Gestor de Ofuscação através da interface *Inicializar* a fim de iniciar toda a operação desse arcabouço. Em seguida, o Gestor de Ofuscação utiliza a interface *SolicitarPerfil* para solicitar ao Gestor de Segurança a definição do Perfil de Segurança adequado para proteger um programa. Após receber o Perfil de Segurança, o Gestor de Ofuscação solicita ao Compilador que o Código Fonte seja manipulado a fim de permitir as ofuscações e em seguida compilado para a Plataforma de Hardware correta (informada pelo usuário). Após a compilação, o Compilador informa ao Gestor de Ofuscação que o arquivo executável Binário foi gerado. O próximo passo requer que o Gestor de Ofuscação solicite ao Ofuscador que aplique as técnicas de ofuscação no Binário. Após realizar essas transformações no Binário, de acordo com o Perfil de Segurança (técnicas de ofuscação a serem empregadas e o grau de ofuscação) definido anteriormente, o Ofuscador gera o Binário Ofuscado e a Lista de Trechos não Executáveis de Código, informando os endereços do Binário Ofuscado onde os dados confidenciais podem ser armazenados de forma segura. Por último, o Gestor de Ofuscação solicita ao Protetor de Dados que os dados listados na Lista de Dados Confidenciais sejam protegidos. Antes de entregar o arquivo executável Binário Ofuscado Protegido, o Protetor de Dados armazena os dados confidenciais nos trechos de código não executáveis e em seguida conserta todas as referências relacionadas a esses dados. Neste momento o programa (Binário Ofuscado Protegido) pode ser entregue ao usuário a fim de implantá-lo nos dispositivos onde esse programa deverá executar.

Estudo de Caso

Nesta seção será apresentado um estudo de caso mostrando o *TinyObf*, instanciação da arquitetura de ofuscação de código e proteção de dados, proposta na seção anterior, para RSSFs. Nesse estudo de caso as técnicas de ofuscação empregadas pelo *TinyObf* foram personalizadas para uma plataforma de hardware utilizada por sensores. A plataforma escolhida foi *MICAz* cuja arquitetura de hardware é *Atmel AVR*. Essa plataforma foi escolhida, pois é uma plataforma bastante representativa, já que possui maiores limitação de recursos em relação à outras plataformas como, por exemplo, SunSpot e Z1 que possuem mais poder de processamento e memória. Portanto os experimentos realizados com o MICAz fatalmente funcionarão para as demais plataformas. Além disso, outro fator que nos fez escolher essa plataforma é que sensores dessa plataforma são amplamente utilizados em pesquisas relacionadas à RSSFs.

Foi desenvolvida uma ferramenta capaz de realizar as transformações no Binário a fim de ofusca-lo.

Durante o desenvolvimento dessa ferramenta, foram identificadas duas situações curiosas: (i) destino das instruções de salto curto inalcançável e (ii) excesso de tamanho do código executável. Vale ressaltar que na primeira situação o código em *assembly* não compilava enquanto na segunda situação o programa apesar de não apresentar erro durante a compilação, não podia ser implantado em um sensor já que seu tamanho ultrapassava o limite de memória disponível.

A primeira situação começou a ocorrer quando tentávamos compilar programas com mais de cem mil instruções *assembly*. Ao adicionarmos instruções *NOP*, constatamos que o destino das instruções de salto curto tornavam-se inalcançáveis, ou seja, além do limite permitido para esse tipo de instrução. Isso foi detectado ao adicionar instruções *NOP* antes das instruções de salto curto relativo, fazendo com que essas instruções não fossem capazes de redirecionar corretamente o fluxo de controle. As instruções de salto curto relativo são capazes de realizar um salto de no máximo 64 endereços na arquitetura *Atmel AVR*, ao adicionar muitos *NOPs* no código, o endereço para onde as instruções deveriam saltar se tornavam inalcançáveis e com isso o fluxo de controle passava a ser redirecionado para endereços indevidos, gerando erros durante a compilação desse programa. Essa situação foi contornada substituindo todas as instruções salto curto relativo por instruções de salto incondicional, já que essas instruções possuem um alcance maior de salto. Para isso usamos uma técnica chamada *branch flipping* (LINN e DEBRAY 2003). Nessa técnica substituem-se as instruções de redirecionamento condicional por duas instruções: (i) instrução de redirecionamento condicional com condição invertida e (ii) uma instrução de salto incondicional. Por exemplo, na arquitetura *Atmel AVR*, substituímos a instrução *BREQ* (*Branch If Equal*), instrução que redireciona o fluxo de controle caso a flag ZERO esteja ativada, pelas instruções *BRNE* (*Branch If Not Equal*), instrução que redireciona o fluxo de controle caso a flag ZERO esteja desativada, e a instrução de salto incondicional *JMP*.

A segunda situação ocorreu porque ao adicionar *NOPs* após as instruções que serão substituídas, o tamanho do programa passou a ultrapassar o limite de memória disponível pela plataforma. Com isso passou a ocorrer erros ao tentar implantar esse programa em um sensor. Apesar dessa situação não apresentar nenhum erro durante a compilação é necessário checar o tamanho do programa a fim de garantir que o tamanho do programa ofuscado não ultrapasse o limite de memória disponível no nó sensor utilizado, neste caso em um sensor da plataforma *MICAz*. Tal problema foi resolvido ao verificar quantos *NOPs* poderiam ser adicionados em um programa. No entanto, apesar de solucionar o problema, tal abordagem acaba limitando o número de ofuscações que podem ser realizadas em um programa.

1.11 Plataforma *MICAz*

A plataforma *MICAz* da família Crossbow possui 4 Kbytes de memória RAM e 128 Kbytes de memória ROM, um microcontrolador *Atmel AVR* de 8 *bits* **(ATMEL 2012)** e um Chipcon 2420 para comunicação sem fio com uma taxa de transmissão de 250*Kbps* a 2,4 *GHz* em redes aderentes ao padrão *IEEE* 802.15.4.

A Figura 19 mostra um sensor *MICAz*, um dispositivo (*MIB520*) utilizado para conectar o *MICAz* ao computador, um sensor *MICAz* associado a *MIB520* e uma placa *mda100* capaz de medir temperatura e luminosidade (essa placa é conectada ao *MICAz* através de um conector de expansão). O *MICAz* pode ser utilizado com o sistema operacional *TinyOS*, sistema operacional open source projetado para dispositivos de baixo consumo de energia e que se comunicam através de redes sem fio (TINYOS 2012). Para desenvolver aplicações *TinyOS* utiliza-se uma linguagem de programação chamada *nesC* (*network embedded system C* – uma

extensão da linguagem C). O *TinyOS* define um modelo de programação baseado em componentes independentes e reutilizáveis. Os componentes podem chamar e receber comandos (utilizados para requisitar serviços, como o envio de uma mensagem, por exemplo), sinalizar e receber eventos (término de um serviço ou a ocorrência de um evento de hardware).

Figura 19. Dispositivos para a plataforma *MICAz*

1.12 Técnicas de Ofuscação Personalizadas para a arquitetura *Atmel AVR*

Nessa seção descrevemos como as quatro técnicas de ofuscação: ofuscação de chamada, retorno, falso retorno e salto incondicional foram personalizadas para a plataforma *MICAz*. Tais técnicas foram personalizadas para a arquitetura *Atmel AVR* já que os sensores utilizados nesse estudo de caso são da plataforma *MICAz* e possuem um processador *ATmega128L* (ATMEL 2012).

É importante ressaltar que para personalizar cada uma dessas técnicas foi necessário estudar as seguintes características da arquitetura de processadores *Atmel AVR:* o seu conjunto de instruções, a sua forma de endereçamento e outras características como, por exemplo, quantos bits podem ser armazenados em um endereço da pilha dessa arquitetura. Portanto caso seja necessário personalizar tais técnicas para outras arquiteturas é necessário empregar um especialista de segurança capaz de aprender tais especificidades da arquitetura em questão.

Ofuscação de Chamada

Como visto na seção 1.4.1, a ofuscação de chamada pode ser realizada substituindo uma instrução que faz uma chamada de função, geralmente instrução *CALL*, por outras instruções capazes de manter a mesma semântica. Para a arquitetura de hardware pretendida (*Atmel AVR*), podemos substituir uma instrução *CALL* por uma combinação de quatro instruções *LDI*, quatro instruções *PUSH* e uma instrução *RET*. Essa combinação empilha o endereço de retorno da função, empilha o endereço da função que será executada e, por último, a instrução *RET* inserida remove o endereço do topo da pilha e transfere o fluxo de controle para esse endereço, ou seja, para o endereço da função que deverá ser executada.

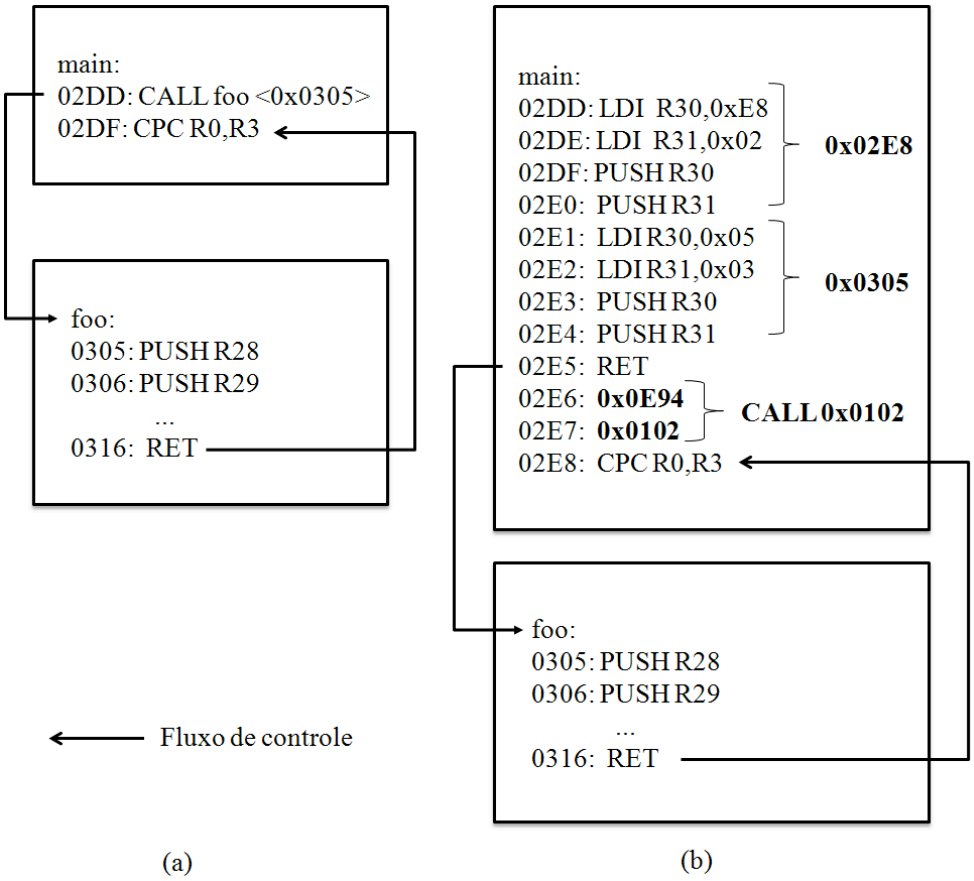

Figura 20. Ofuscação de chamada para a plataforma MICAz

A Figura 20(a) apresenta como o fluxo de controle flui entre as funções main e foo. Já a Figura 20(b) mostra o mesmo fluxo de controle, mas após a chamada de função localizada no endereço 0x022D ter sido ofuscada com a ofuscação de chamada. A pilha usada pelo *ATmega128L* é capaz de armazenar somente dados de 8 bits provenientes de um dos registradores de uso geral (*R16-R31*). No entanto para representar um endereço de memória dessa arquitetura são necessários 16 *bits*. Com isso, a carga de um endereço na pilha deve ser feita em dois estágios. No primeiro estágio empilha-se a parte baixa do endereço (bits de 0 a 7) e no segundo estágio, empilha-se a parte alta do endereço (bits de 8 a 15). No entanto, antes disso é necessário atribuir as constantes que representam a parte baixa e alta de um endereço nos registradores de uso geral. Neste caso utilizamos a instrução *LDI* (*Load Immediate*), instrução que copia o valor de uma constante de 8 bits para um registrador de uso geral. No exemplo utilizado utilizamos os registradores R30 e R31, pois são próprios para a carga de endereços na pilha.

Na Figura 20. Ofuscação de chamada para a plataforma MICAz(a) a função foo é chamada no endereço *0x02DD* e após a execução dessa função o fluxo de controle é redirecionado para o endereço *0x02DF* onde está localizada a instrução *CPC R0,R3*. É importante ressaltar que foi necessário realocar a instrução *CPC R0,R3* a fim de realizar a ofuscação, por isso o endereço de retorno de função passou a ser 0x02E8. Para realizar a ofuscação de chamada de *foo* foram necessários 4 instruções *LDI* e 4 instruções *PUSH* como visto na Figura 20. Ofuscação de chamada para a plataforma MICAz(b). Duas instruções *LDI* e duas instruções *PUSH* são utilizadas para carregar um endereço de memória na pilha. Portanto, primeiro empilha-se o endereço de retorno da função (0x02E8) e em seguida empilha-se o endereço da função a ser executada (0x0305).

Durante a execução desse código, quando a primeira instrução RET, localizada no endereço 0x02E5 for executada, o fluxo de controle será redirecionado para o primeiro endereço da função *foo* e quando a instrução *RET* de *foo*, localizada no endereço 0x0316, for executada, o fluxo de controle será redirecionado de volta para a função Main no endereço *0x0305* mantendo a mesma semântica do programa original.

No exemplo mostrado na Figura 20(b) foi criado um trecho de código não executável, localizado entre os endereços 0x02E6 e 0x02E7. Esse trecho de código não executável foi utilizado para proteger uma chave criptográfica de quatro *bytes 0x0E940102* que pode ser confundido com a instrução *CALL 0x0102* quando uma ferramentas de engenharia reversa tenta traduzir os bytes desse código.

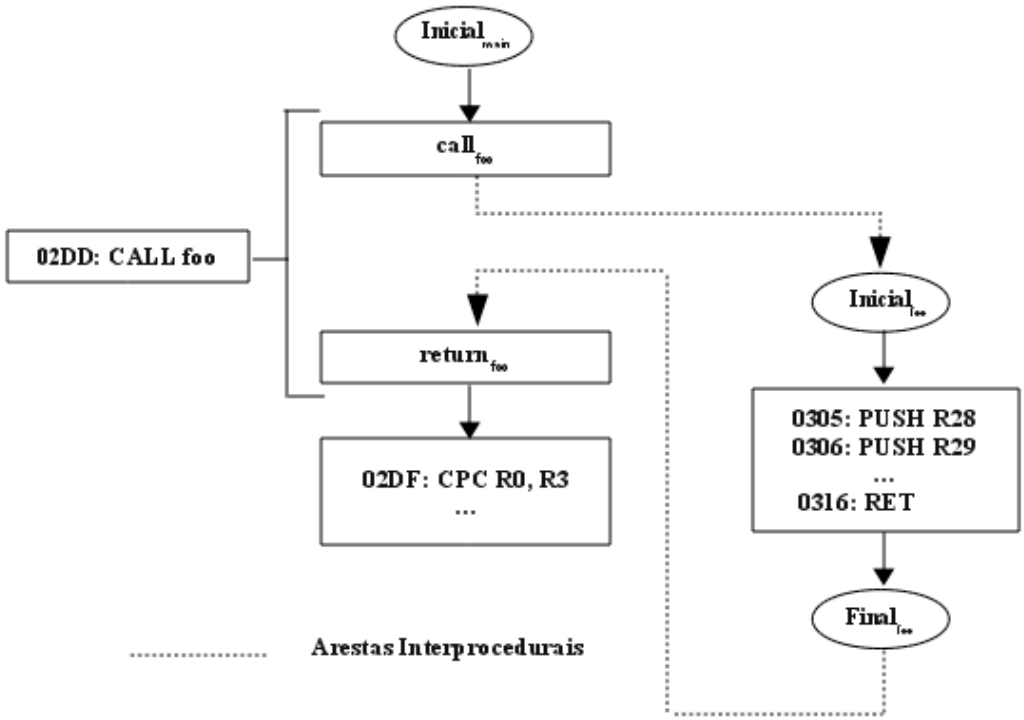

Figura 21. ICFG do programa original mostrado na Figura 20(a)

A Figura 21 mostra o ICFG referente ao programa da Figura 20(a). Nesta figura, podemos ver que no endereço onde é realizada a chamada de função, endereço 0x02DD, há uma aresta interprocedural conectando a função main através do nó de chamada, denominado $call_{foo}$, com o nó inicial do CFG da função foo ($Inicial_{foo}$). Além dessa, existe outra aresta interprocedural entre a função main e a função foo, aresta que conecta o nó final de foo, denominado $final_{foo}$ com o nó de retorno de main, denominado $return_{foo}$. Através desse grafo é possível ver claramente o relacionamento entre as funções main e foo.

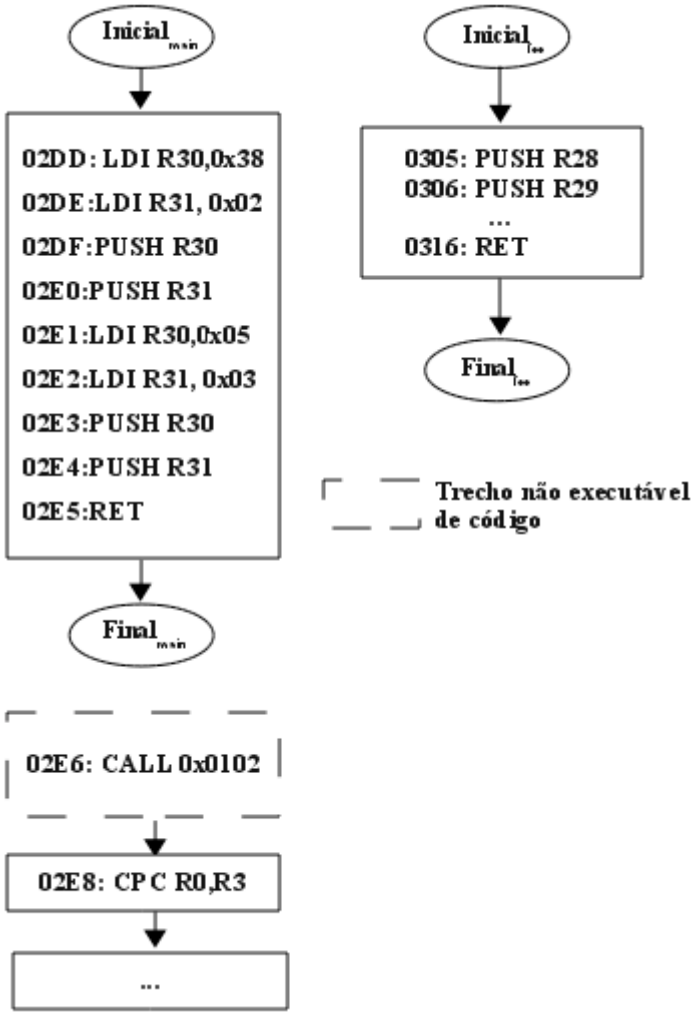

Figura 22. ICFG do programa ofuscado mostrado na Figura 20(b)

A Figura 22 mostra como o ICFG fica corrompido em relação ao grafo original (Figura 21) ao utilizar a técnica de ofuscação de chamada. O grafo do ICFG ofuscado mostra que as arestas interprocedurais que conectavam main e foo foram removidas e, com isso, o relacionamento entre essas funções não fica mais evidente. Além disso, por causa da instrução RET inserida no endereço 0x02E5, o fim do CFG da função main é identificado incorretamente, pois outras instruções, inclusive a instrução "CPC R0,R3" localizada no endereço 0x02E8 também fazem parte dessa função. Em relação à camuflagem de dados, após armazenar a chave criptográfica 0x0102 concatenada com os bytes que identificam o operador de uma instrução CALL 0x0E94, a instrução "CALL 0x0102" localizada no endereço 02E6 foi identificada como sendo a instrução que será executada antes da instrução "CPC R0,R3". No entanto essa chamada de função nunca será executada por conta da manipulação do fluxo de controle. Por outro lado, essa instrução não deveria mesmo ser executada já que seus bytes identificam um dado confidencial, chave criptográfica, e não uma instrução.

Ofuscação de Retorno

Como visto na seção 1.4.2, uma instrução *RET* pode ser substituída por um conjunto de instruções capazes de realizar o retorno de uma função, ou seja, capazes de redirecionar o fluxo de controle de volta para o ponto onde a função foi chamada. Para a arquitetura de hardware *Atmel AVR*, podemos substituir uma instrução *RET* por

uma combinação de duas instruções *POP* e uma instrução *IJMP*.

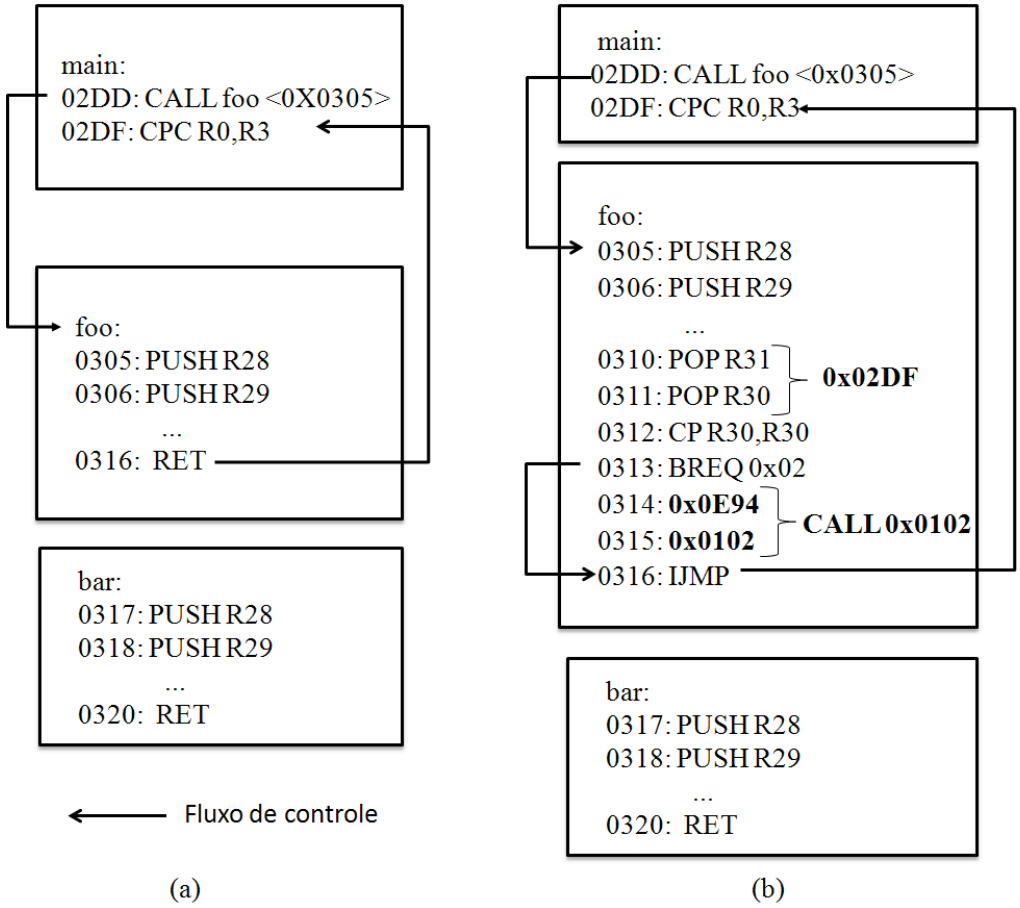

Figura 23. Ofuscação de retorno para a plataforma MICAz

O exemplo mostrado na Figura 23 apresenta a ofuscação de retorno considerando o conjunto de instruções do processador *ATmega128L*. Nesse exemplo, a ofuscação de retorno, mostrada na Figura 23(b) utiliza duas instruções POP, localizadas nos endereços 0x0310 e 0x0311 para copiar a parte alta e baixa do endereço de retorno localizados no topo da pilha para os registradores R30 e R31. Em seguida utilizamos um predicado opaco (*CP* R30, R30) e uma instrução de redirecionamento condicional (*BREQ*) a fim de criar um trecho não executável de código a fim de camuflar a chave criptográfica 0x*0102*. O predicado opaco "*CP R30, R30"* (endereço 0x0312), instrução de comparação que sempre ativa a flag ZERO independente do valor armazenado em *R30*, é utilizado para garantir que o fluxo de controle sempre será redirecionado para o endereço onde está armazenada a instrução de redirecionamento condicional "*BREQ 0x02"*. Esta instrução, por sua vez, redireciona o fluxo de controle para o endereço 0x0316 onde está localizada a instrução de redirecionamento incondicional '*IJMP'*, responsável por fazer o retorno da função. Neste exemplo, podemos notar que o redirecionamento realizado pela instrução "*BREQ* 0x02" cria um trecho de código não executável compreendido entre os endereços *0x0314* e *0x0315,* capaz de armazenar um dado confidencial. Neste caso, esse trecho foi utilizado para armazenar a chave criptográfica *0x0E940102,* que por sua vez, é traduzida como se fosse a instrução *CALL0 x0102*.

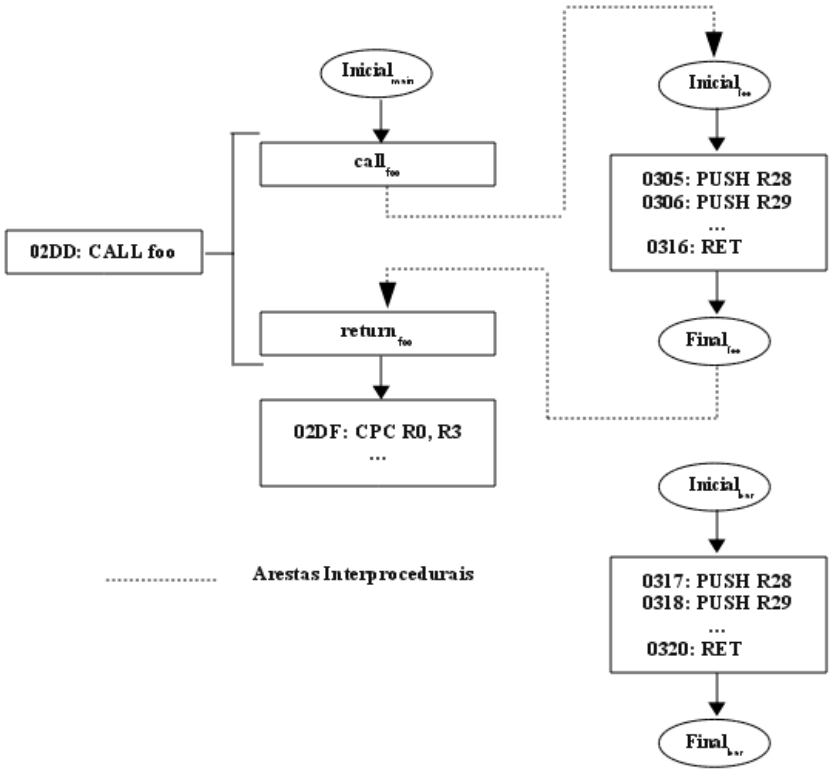

Figura 24. ICFG do programa original mostrado na Figura 23(a)

A Figura 24 mostra o *ICFG* do programa não ofuscado da Figura 23(a). Esta figura é semelhante a Figura 21(a). A única diferença é que foi adicionado o *CFG* da função bar iniciado no endereço 0x0317 e que possui os nós de início e final denominados $Inicial_{bar}$ e $Final_{bar}$ respectivamente. A ofuscação de retorno induziu a geração incorreta do ICFG, como mostrado na Figura 25. Através desse ICFG, podemos perceber que a aresta interprocedural que conectava $Final_{foo}$ com $return_{foo}$ foi removida e em seu lugar foi inserida uma aresta partindo de $Final_{bar}$ para $return_{foo}$. Com isso, percebe-se que o final da função foo foi confundido com o final da função bar já que a instrução RET da função foo foi removida e por isso a próxima instrução RET, instrução de retorno da função bar localizada no endereço 0320, foi identificada como o fim da função foo.

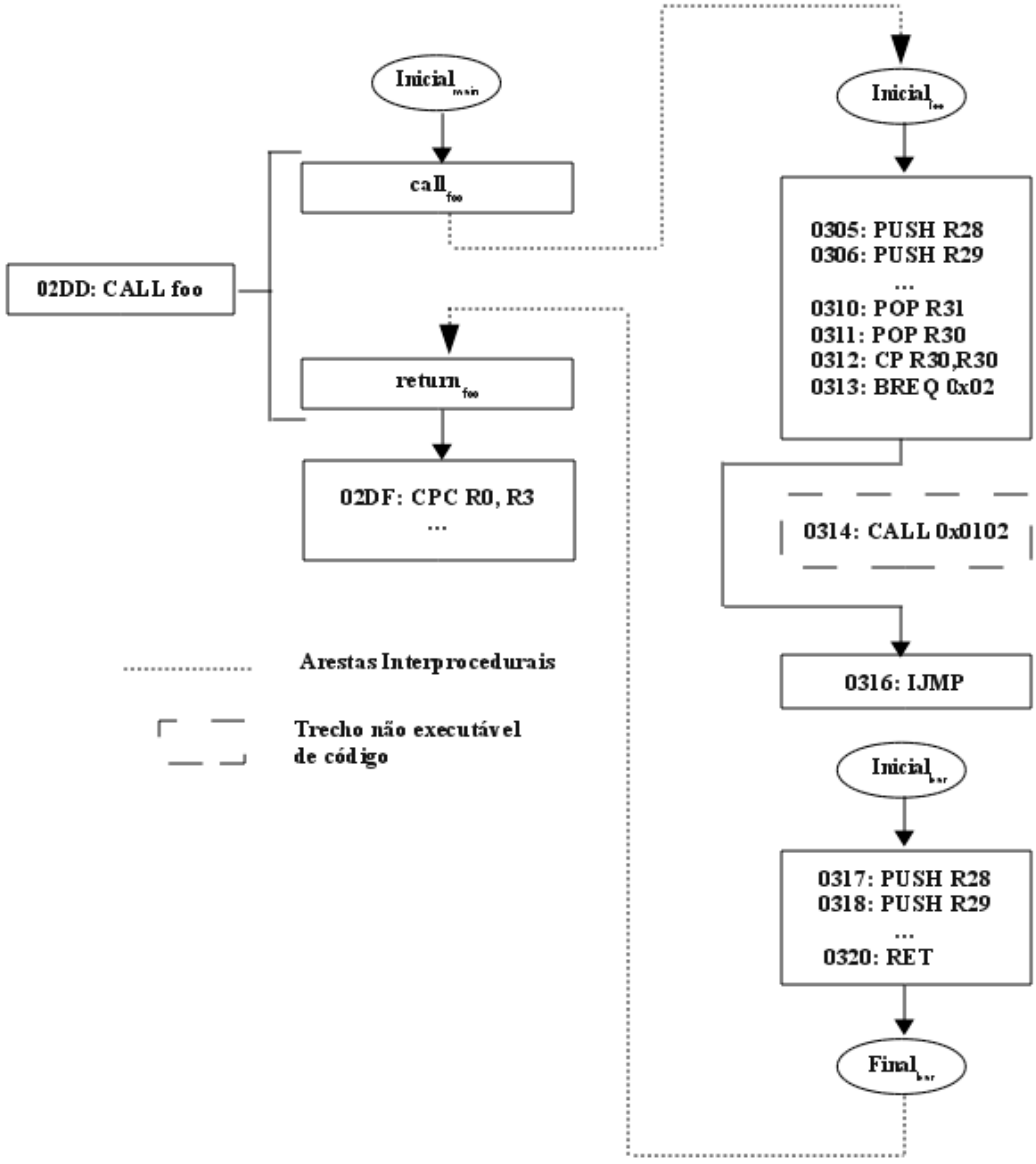

Figura 25. ICFG do programa ofuscado mostrado na Figura 23(b)

A chamada de função "CALL 0x0102", localizada no endereço 0x0314, nunca será executada por causa do predicado opaco localizado no endereço 0x0313. Além disso, essa instrução é, na verdade, a chave criptográfica que está sendo camuflada dentro do código do programa. Dessa forma, um atacante deverá gastar mais tempo e esforço para descobrir essa chave criptográfica.

Ofuscação de Falso Retorno

Como visto na seção 1.4.3, a ofuscação de falso retorno pode ser realizada adicionando uma instrução *RET* que não caracteriza o fim legítimo de uma função. A Figura 26(a) apresenta um código que não possui um falso retorno. Neste código, o retorno da função foo redireciona o fluxo de controle do programa para o endereço 0x02DF onde está localizada a instrução "*CPC R0, R3"*. A Figura 26(b), por sua vez, mostra como a função foo pode ser ofuscada através do uso do falso retorno. Para isso, foram utilizadas duas instruções *LDI* e duas instruções *PUSH* para armazenar no topo da pilha o endereço de retorno relativo ao falso retorno, endereço

0x030E onde está localizada a instrução "*MOVW* R16,R24", a fim de redirecionar o fluxo de controle para a próxima instrução dentro da própria função foo. Por ultimo, o falso retorno, instrução *RET* localizada no endereço 0x030B, é inserida a fim de comprometer as ferramentas de engenharia reversa interessadas em analisar a função foo, já que estas irão deduzir que o fim da função *foo* está localizado no endereço 0x030B quando na verdade o seu fim legítimo é no endereço 0x0316.

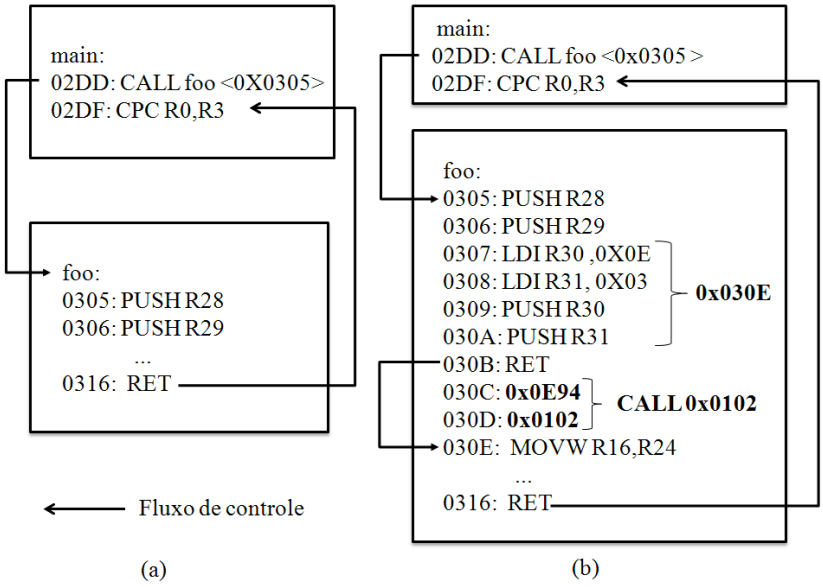

Figura 26. Ofuscação de falso retorno para a plataforma MICAz

Ainda de acordo com o exemplo mostrado na Figura 26(b), podemos notar que foi criado um trecho não executável de código entre os endereços 0x030C e 0x030D que foram utilizados para armazenar a chave criptográfica *0x0E940102*. Esse trecho foi criado através da manipulação do endereço de retorno utilizado pelo falso retorno.

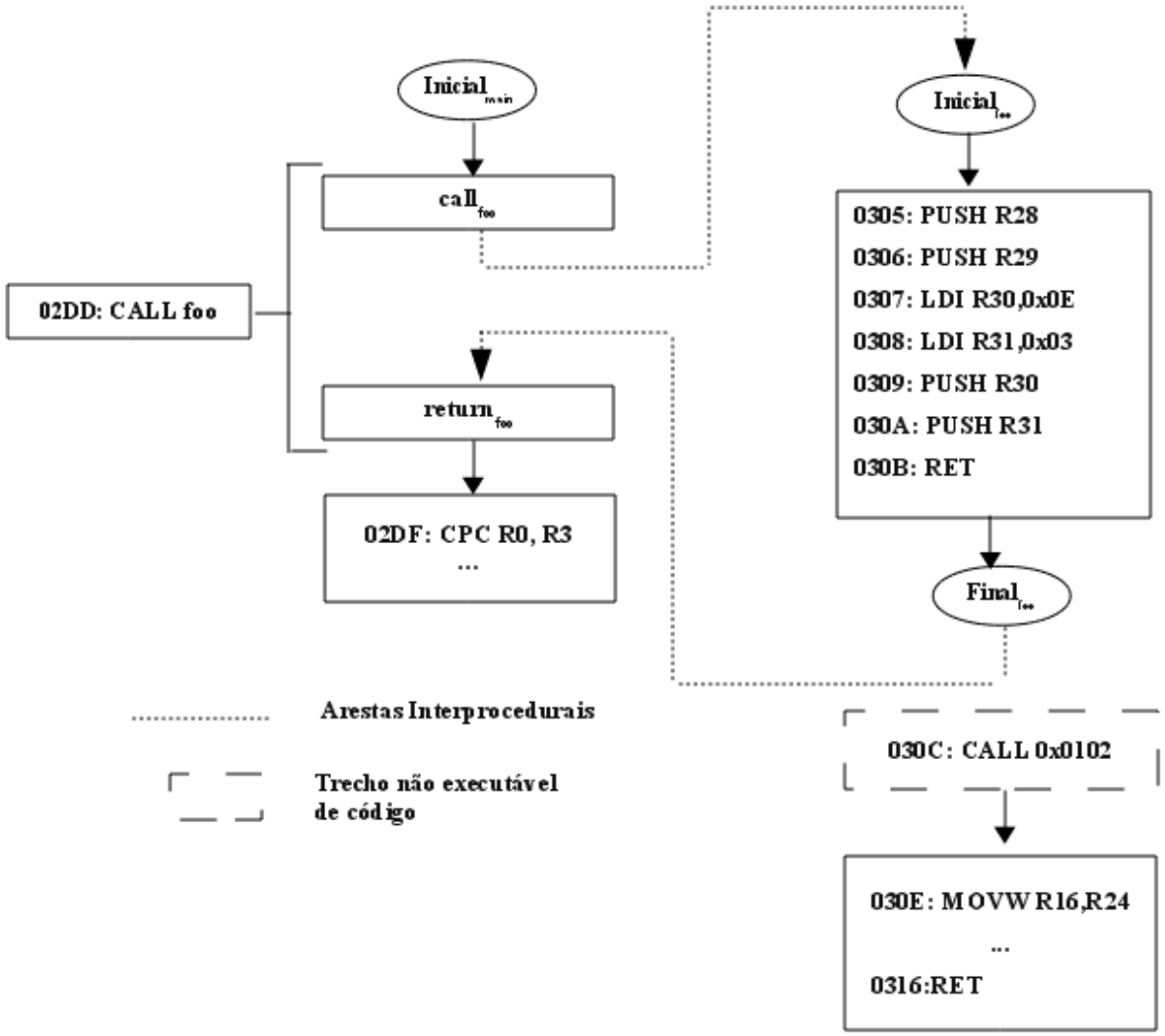

Figura 27. ICFG do programa ofuscado mostrado na Figura 26(b)

A Figura 27 mostra como o nó final de foo ($Final_{foo}$) foi identificado incorretamente, ou seja, após a instrução *RET* localizada no endereço 0x030B. Com isso a aresta interprocedural que demonstra o relacionamento entre e main e foo foi gerada de forma incorreta. Uma das consequências disso é que a análise do *CFG* de foo poderá ser feita de maneira incorreta já que não irá considerar todas as instruções legítimas dessa função. Assim como nos outros exemplos, no código ofuscado, também foi criado um trecho de código não executável, localizado nos endereços 0x030C, que foram utilizados para armazenar uma chave criptográfica que induz as ferramentas de engenharia reversa a traduzirem essa chave como se fosse uma instrução.

Ofuscação de Salto Incondicional

Como visto na seção 0, a ofuscação de salto incondicional pode ser realizada substituindo-se as instruções *JMP* por chamadas de função para um tipo especial de função denominada função de redirecionamento. A Figura 28 apresenta um exemplo de ofuscação de salto incondicional onde a instrução *JMP* localizada no endereço 0x02DD é substituída por uma chamada de função, "CALL bf". A função que está sendo chamada (bf) é desenvolvida de tal maneira que seu comportamento seja o mesmo de uma instrução *JMP*, ou seja, simplesmente redireciona o fluxo de controle para um determinado endereço passado como parâmetro, ou seja, endereço de

destino da instrução *JMP*.

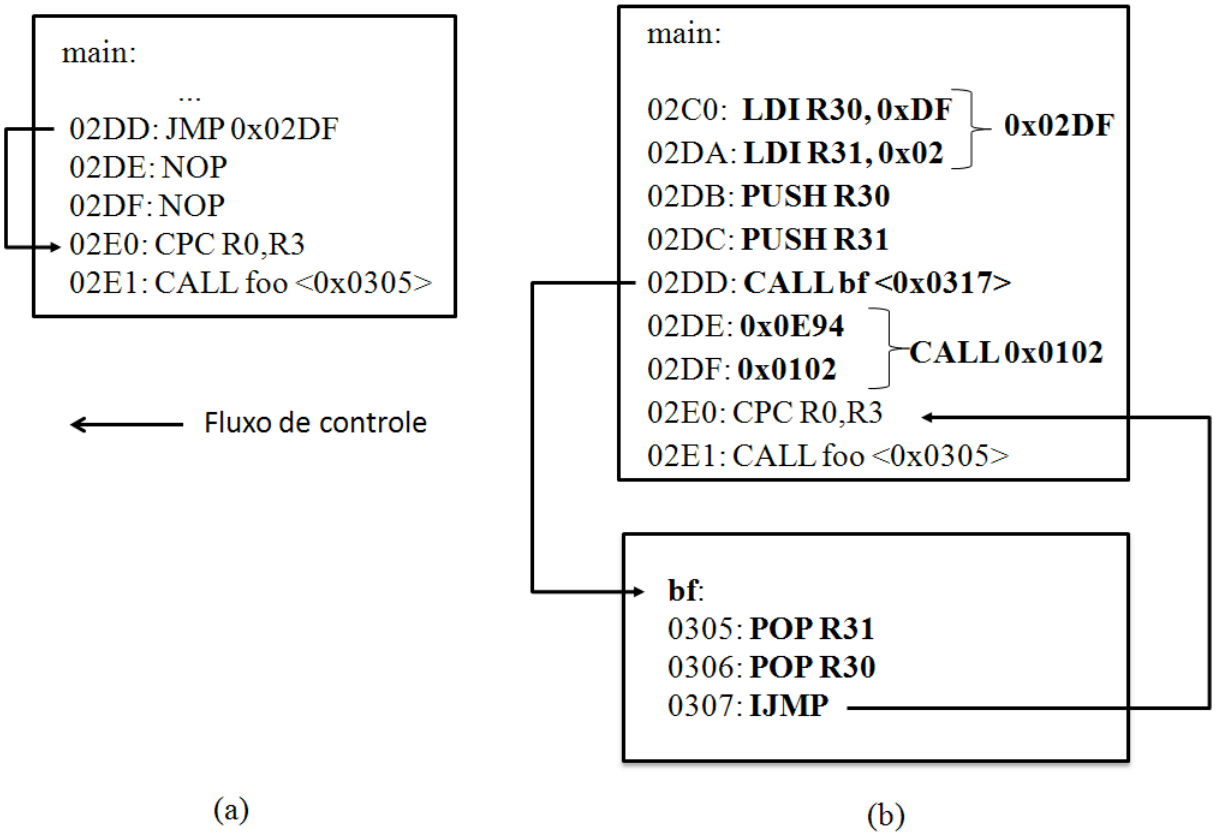

Figura 28. Ofuscação de salto incondicional para a plataforma MICAz

As instruções compreendidas entre os endereços 0x02C0 e 0x02DC mostrados na Figura 28(b) fornecem o parâmetro da função chamada a seguir, ou seja, bf. Para que bf se comporte como uma instrução *JMP* é necessário que o fluxo de controle seja redirecionado para o endereço armazenado na pilha (parâmetro da função). Para isso bf é composto de duas instruções *POP*, responsáveis por armazenar nos registradores R30 e R31 a parte baixa e alta do destino da instrução de redirecionamento e uma instrução *IJMP* que simplesmente redireciona o fluxo de controle para o endereço composto pelos registradores R30 e R31. Neste exemplo, o fluxo de controle ainda foi manipulado com o intuito de criar um trecho de código não executável, endereços compreendidos entre 0x02D3 e 0x02DF a fim de proteger a chave criptográfica 0x0E940102.

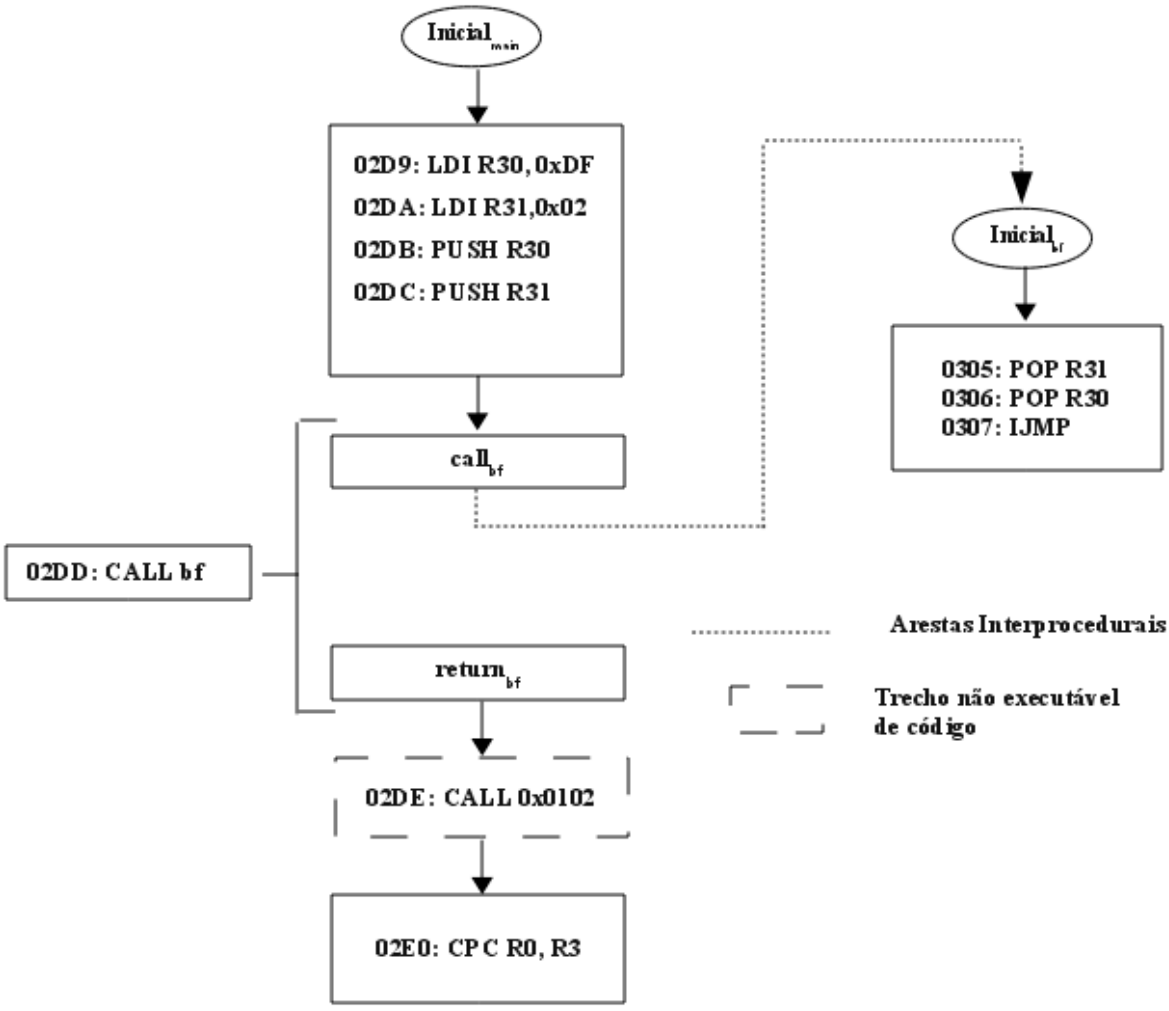

Figura 29. ICFG do programa ofuscado mostrado na Figura 28(b)

A Figura 29 mostra como ficou o *ICFG* do programa ofuscado mostrado na Figura 28(b). Nesta figura podemos ver que não há uma aresta interprocedural de retorno conectando o nó final da função bf com a função main. Isso se explica porque em bf não há uma instrução de retorno. Quando a instrução *IJMP* localizado no endereço 0x0307 é executada o fluxo de contole é redirecionado para o endereço 0x02E0, endereço armazenado no topo da pilha. No entanto durante a geração do *ICFG* após o nó de retorno $return_{bf}$ a próxima instrução a ser executada é a instrução localizada no endereço 0x02DE: CALL 0x0102 que na verdade nunca será executada por conta da manipulação do fluxo de controle já que essa instrução é na verdade a chave criptográfica que está sendo protegida.

Experimentos e Análise de Resultado

Esta seção detalha os experimentos realizados para avaliar a (i) eficácia; a (ii) eficiência e a (iii) furtividade do TinyObf. Além disso, essa seção ainda conta com uma discussão sobre os benefícios do mecanismo de proteção empregado por esse arcabouço e outra que apresenta argumentos que afirmam porque esse arcabouço pode ser considerado resistente contra desofuscadores. A eficácia do TinyObf é medida para indicar o aumento da dificuldade em comprometer um sensor ofuscado através da engenharia reversa. A eficiência é medida para calcular o impacto do TinyObf em relação ao consumo de recursos dos sensores ao ofuscar o programa

embarcado neles. Por ultimo a furtividade é medida para mostrar que as ofuscações aplicadas pelo TinyObf não fornecem indícios de que o programa embarcado no sensores está ofuscado, característica que torna o desenvolvimento de desofuscadores automáticos mais difícil.

1.13 Ambiente dos Experimentos

A RSSF utilizada nos experimentos é composta por sensores da plataforma *MICAz* compostos por microcontroladores de 8 bits da arquitetura *Atmel AVR*. Os programas utilizados foram desenvolvidos com *TinyOS* 2.1.1 utilizando a linguagem *nesC*.

A **eficácia** do *TinyObf* foi avaliada a partir de informações fornecidas pelo *avr-objdump* (SAVANNAH 2012) e pelo *IDA PRO* (HEX-RAYS 2012). Ambos foram escolhidos, pois implementam os principais algoritmos de desmontagem. O *avr-objdump*, pois utiliza o algoritmo varredura linear e o *IDA PRO*, pois utiliza algoritmo transversal recursivo.

Os experimentos relacionados à ***furtividade*** do *TinyObf* também foram realizados utilizando informações obtidas do *avr-objdump* e do *IDA PRO*. Tal aspecto é medido comparando os programas ofuscados com um conjunto de mais de 30 programas não ofuscados provenientes do ambiente de programação do TinyOS (TINYOS 2012). É bom ressaltar que tanto a eficácia quanto a furtividade foram avaliadas antes de o programa ser implantado nos sensores.

Os experimentos relacionados à ***eficiência*** do *TinyObf* foram realizados através de simulação usando o *AvroraZ* (ALBEROLA e PESCH 2008). O simulador Avrora na versão 2.6 é um simulador de código aberto para RSSF. Especificamente, a extensão AvroraZ foi empregada para avaliar o consumo de energia e de ciclos de processamento. O modelo de energia usado é chamado de "*Accurate Prediction of Power Consumption in Sensor Networks*" (AEON) (LANDSIEDEL *et al.* 2005) e é o modelo de energia que representa com maior precisão os ciclos de processamento do sensor.

1.14 Descrição das Métricas

Muitos trabalhos da literatura (LINN e DEBRAY 2003); (POPOV *et al*. 2007); (LEE 2010) utilizam o fator de confusão para medir a **eficácia** da ofuscação. O ***fator de confusão (CF)*** é usado para medir o aumento da dificuldade em analisar um programa ofuscado em relação a sua versão original, ou seja, sua versão não ofuscada.

A Equação **Error! Bookmark not defined.** define *CF*, onde $|A|$ é o conjunto de endereços das instruções reais do programa, P é o conjunto de endereços das instruções percebidas pelas ferramentas de engenharia reversa e $|A - P|$ é o conjunto de endereços das instruções que não foram corretamente identificadas pelas ferramentas de engenharia reversa. Portanto à medida que o valor de *CF* aumenta, significa que as ferramentas de engenharia reversa não foram capazes de traduzir corretamente as instruções do programa e, com isso, a análise do programa ofuscado se torna mais difícil.

$$CF = \frac{|A - P|}{|A|}$$

Equação **Error! Bookmark not defined.**

A **furtividade** mede o quanto um programa ofuscado se assemelha com um programa não ofuscado. Quanto maior for essa semelhança mais difícil será de identificar que um programa está ofuscado dificultando o processo de desofuscação. A furtividade pode ser avaliada utilizando a **distância de *mahalanobis* (md)** (LEE 2010), A distância de mahalanobis calcula a diferença entre dois vetores, compostos de variáveis aleatórias escalares do mesmo espaço de probabilidade, e informa a dissimilaridade entre eles. Neste trabalho, a distância de mahalanobis foi utilizada para medir a dissimilaridade entre a distribuição de instruções de um programa ofuscado em relação à distribuição de instruções de um conjunto de programas não ofuscados.

$$md(\vec{N}, \vec{X}) = \sqrt{(\vec{N} - \vec{X})^T S^{-1} (\vec{N} - \vec{X})}$$

Equação **Error! Bookmark not defined.**

A Equação **Error! Bookmark not defined.** apresenta a expressão utilizada para calcular md. Cada elemento do vetor *N* é uma função massa de probabilidade (pmf) dos *opcodes* encontrados em um programa não ofuscado, *X* é o vetor cujos componentes apresentam a ocorrência dos *opcodes* de um programa ofuscado e *S* a matriz diagonal cujos elementos da diagonal principal são as variâncias do conjunto de programas utilizados para criar o vetor *N*. Portanto, quanto maior a distância de mahalanobis, mais a distribuição de um programa ofuscado difere em relação aos programas não ofuscados. Portanto caso a distância de mahalanobis calculada para um programa ofuscado seja semelhante a distância do programa não ofuscado significa que a ofuscação não dá indícios que o programa está ofuscado e portanto pode ser considerada como furtiva.

A **eficiência** do *TinyObf* foi avaliada em termos do seu impacto em relação ao consumo de recursos dos sensores tais como o tamanho do programa, ciclos de processamento e consumo de energia. O **tamanho de um programa** é o número de bytes necessários para compor o conjunto de instruções e dados de um programa. O número de **ciclos de processamento** mostra quanto de CPU o programa está utilizando. O **consumo de energia** está associado ao gasto de energia da CPU.

1.15 Resultados do *TinyObf*

Nesta seção são apresentados os resultados obtidos após a realização dos experimentos com os seguintes programas: *BaseStation*, *Blink*, *RadioCountToLeds*, *RadioSenseToLeds* e *Sense*. **BaseStation** é um aplicação que troca mensagens entre uma estação base conectada pela porta serial e os nós sensores através do enlace sem fio. **Blink** é uma aplicação que alterna o uso dos *leds* do sensor de acordo com um contador interno. **RadioCountToLeds** é uma aplicação que envia periodicamente uma mensagem em *broadcast* contendo o valor de um contador interno de 4Hz. Essa aplicação é útil para testar a comunicação entre os nós de uma RSSF já que o sensor que recebeu essa mensagem acende seus *leds* de acordo com o valor recebido de um outro sensor. **RadioSenseToLeds** é uma aplicação que monitora a luminosidade do ambiente a cada 250ms e envia o dado coletado em broadcast. Outro sensor que também esteja executando essa aplicação, ao receber esse dado pela interface de rede, acende seus leds de acordo com o valor recebido. Por fim, **Sense** é uma aplicação que alterna o

uso dos seus *leds* de acordo com o valor coletado pela unidade de sensoriamento (essa aplicação é útil para testar o funcionamento da unidade de sensoriamento de um nó sensor).

Eficácia

Nessa seção são apresentados os resultados obtidos do cálculo do fator de confusão (CF). CF foi calculado em relação às instruções (instr) e funções (funcs) identificadas pelos disassemblers (*avr-objdump* e *IDA PRO)*. Conforme mencionado, esse fator mede o aumento da dificuldade em realizar a engenharia reversa de um programa em relação ao programa original (programa não ofuscado). Portanto quanto maior a porcentagem do fator de confusão, mais difícil é comprometer um nó sensor.

A Tabela 1 mostra os resultados obtidos para os programas escolhidos para realizar esse experimento. Cada programa foi ofuscado aplicando todas as técnicas de ofuscação implementadas pelo *TinyObf* (ofuscação de chamada, retorno, falso retorno e salto incondicional) com o grau de ofuscação máximo, ou seja, 100%. A Tabela 1 é organizada da seguinte maneira: as linhas representam os programas e as colunas os fatores de confusão tanto para funções (funcs) quanto para instruções (instrs) para cada uma das ferramentas utilizadas (*avr-objdump* e *IDA PRO*). De acordo com os dados dessa tabela o uso da ofuscação tornou a análise do programa ofuscado mais difícil já que houve um aumento percentual no fator de confusão para todos os programas ofuscados.

Os resultados do fator de confusão para as instruções (intrs) mostram que as ofuscações foram capazes de comprometer a desmontagem dos programas ofuscados, pois o aumento do fator de confusão demonstra que a desmontagem desses programas gerou códigos incorretos. Já o aumento percentual do fator de confusão para as funções mostra que as ofuscações podem acarretar a geração de ICFG, CG e CFGs não confiáveis já que as funções dos programas ofuscados não foram identificadas corretamente.

Tabela 1. Fator de Confusão (CF).

Programas	$CF_{avr\text{-}objdump}$		$CF_{IDA\ PRO}$	
	instrs	funcs	instrs	Funcs
BaseStation	53.22%	73.41%	42.30%	80.86%
Blink	50.13%	61.72%	39.53%	75.88%
RadioCountToLeds	56.50%	77.95%	43.65%	81.07%
RadioSenseToLeds	55.26%	76.58%	42.20%	74.73%
Sense	51.83%	63.84%	41.02%	72.03%

Furtividade

Nessa seção são apresentados os resultados obtidos dos experimentos realizados para avaliar a furtividade do *TinyObf*. Assim como a eficácia, a furtividade do *TinyObf* foi avaliada aplicando todas as técnicas de ofuscações (ofuscação de chamada, retorno, falso retorno e salto incondicional) com o grau de ofuscação máximo, ou seja, 100% em cada um dos cinco programas escolhidos (BaseStation, Blink, RadioCountToLeds, RadioSenseToLeds

e Sense). Conforme já foi mencionado, a distância de *mahalanobis* (md) mede a dissimilaridade entre programas. furtividade pode ser avaliada utilizando a distância de mahalanobis para calcular a dissimilaridade entre um programa ofuscado e de um não ofuscado. Neste experimento utilizamos trinta programas do TinyOS para compor a nossa base de dados de programas não ofuscados. A Caso o valor obtido da distância de mahalanobis para o programa ofuscado seja semelhante ao valor obtido da distância de mahalanobis para o programa não ofuscado, podemos considerar que a ofuscação é furtiva.

Tabela 2. Distância de Mahalanobis (md)

Programas	md_{nonobf}	md_{obf}	$md_{diferença}$
BaseStation	19.10	17.85	7%
Blink	5.19	5.34	3%
RadioCountToLeds	26.01	28.34	9%
RadioSenseToLeds	26.01	28.34	9%
Sense	5.21	5.34	2%

A Tabela 2 apresenta os valores de md para os programas não ofuscados (nonobf) e ofuscados (obf). Nessa tabela, as linhas identificam os programas, a primeira coluna identifica o valor de md para um programa não ofuscado (md_{nonobf}), a segunda coluna o valor de md para um programa ofuscado (md_{obf}) e, por fim, a terceira coluna, contém a diferença entre os valores de md da primeira e da segunda coluna de modo a ilustrar o aumento percentual da dissimilaridade entre o programa ofuscado em relação ao não ofuscado ($md_{diferença}$). Através dos resultados apresentados nessa tabela, pode-se observar que os aumentos percentuais entre a distância de *mahalanobis* dos programas ofuscados em relação aos respectivos não ofuscados, vide coluna $md_{diferença}$, são muito pequenos. Portanto, pode-se considerar que os programas ofuscados são semelhantes aos programas não ofuscados. Assim podemos considerar que o *TinyObf* é furtivo, já que os cinco programas ofuscados não dão indícios suficientes para serem identificados como programas ofuscados. Logo as ofuscações empregadas pelo *TinyObf* podem ser consideradas furtivas por natureza, dificultando a detecção de um programa ofuscado com esse arcabouço.

Eficiência

Nessa seção são apresentados os resultados em relação à eficiência do *TinyObf* em termos do consumo de recursos dos sensores. Esse consumo foi avaliado em termos do tamanho do programa, número de ciclos de processamento e consumo de energia dos cinco programas escolhidos para serem ofuscados com o *TinyObf* (BaseStation, Blink, RadioCountToLeds, RadioSenseToLeds e Sense). Tais programas foram ofuscados aplicando todas as técnicas de ofuscação (ofuscação de chamada, retorno, falso retorno e salto incondicional) utilizando os seguintes graus de ofuscação: 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% e 100%. Observe que no grau 0%, o valor mostrado na tabela diz respeito ao programa não ofuscado.

O tamanho do programa é uma medida que não necessita da simulação do programa. A Figura 30 apresenta os valores obtidos em relação ao tamanho dos cinco programas ofuscados pelo *TinyObf*. No eixo das ordenadas (y) são apresentados os tamanhos dos programas em Bytes e no eixo das abscissas (x) são apresentados os diferentes graus de ofuscação. Cada curva nessa figura representa o tamanho de um programa

ofuscado de acordo com os graus de ofuscação. Tais curvas mostram o impacto do tamanho do programa à medida que o grau de ofuscação aumenta. Em cada um dos programas ofuscados, ao aumentar o grau de ofuscação de todas as técnicas de ofuscação aplicadas, ocorre um aumento de aproximadamente 0.0258%, 0.0447%, 0.0637%, 0.0827%, 0.0999%, 0.1019%, 0.1036%, 0.1005%, 0.1017% e 0.1961% em relação ao tamanho do programa não ofuscado (grau de ofuscação de 0%) para respectivamente os graus de ofuscação de 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% e 100%. De acordo com o aumento obtido à medida que o grau de ofuscação aumenta, pode-se considerar que as ofuscações não impactaram negativamente no tamanho dos programas, pois o aumento percentual é muito pequeno. Portanto as ofuscações aplicadas pelo *TinyObf* podem ser consideradas eficientes em relação ao tamanho do programa. Tal resultado comprova a afirmação efetuada nesse trabalho que somente a substituição e inserção de poucas instruções não impactam negativamente no tamanho do programa.

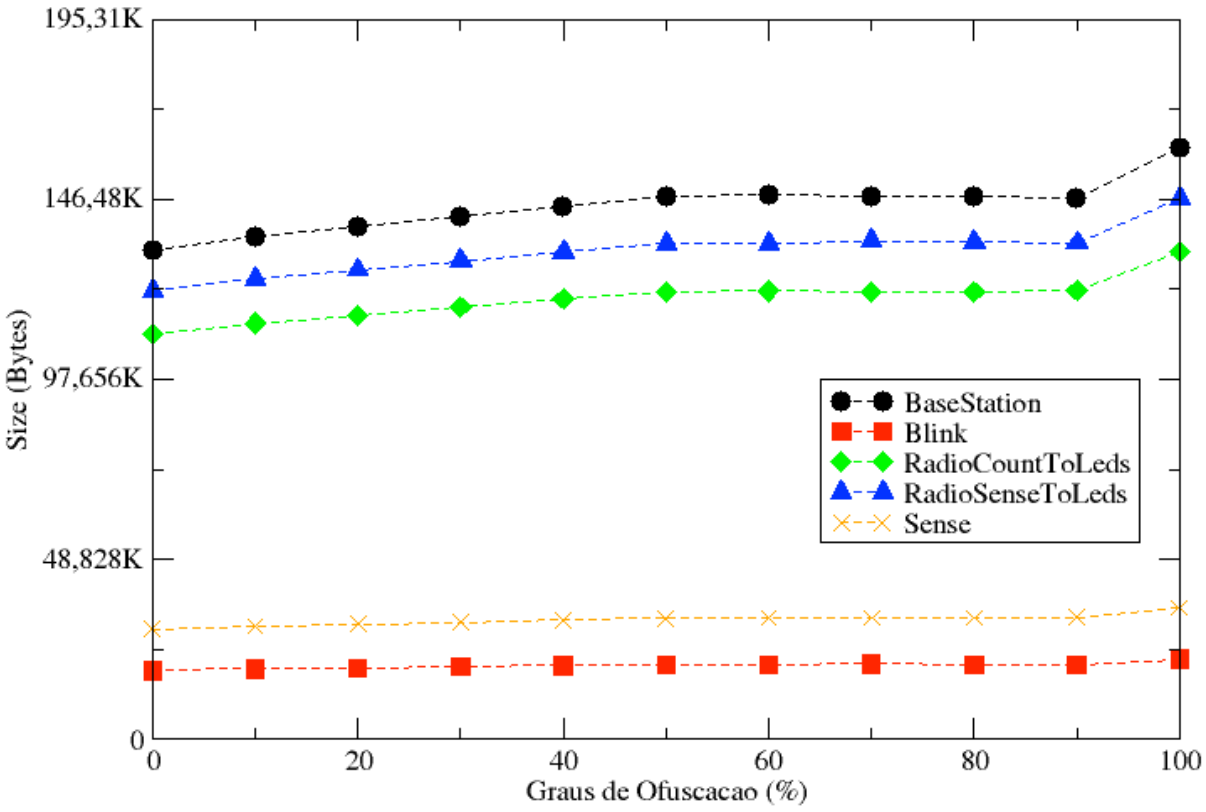

Figura 30. Tamanho do Programa (Bytes).

Os ciclos de processamento e o consumo de energia foram avaliados através de simulação utilizando o AvroraZ. A coleta de resultados foi feita a partir de uma única simulação para cada programa nos seguintes graus de ofuscação: 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% e 100%. Essa simulação não foi repetida, pois os valores obtidos da simulação são determinísticos, ou seja, são sempre os mesmos independente do número de repetições. Isso se deve porque o mesmo conjunto de instruções é executado durante uma rodada de simulação.

Cada instrução de um programa necessita de um determinado número de ciclos de processamento para desempenhar uma determinada operação. Ao ofuscar um programa em utilizando um determinado grau de ofuscação, um determinado número de instruções são substituídas e adicionadas ao programa. Com isso espera-

se que os ciclos de processamento e o consumo de energia aumentem à medida que o grau de ofuscação aumenta. No entanto esse aumento deve ser pequeno visto que a ofuscação adicionam poucas instruções.

A Figura 31 apresenta o número de ciclos de processamento obtidos durante a simulação dos cinco programas escolhidos para serem ofuscados pelo *TinyObf*. No gráfico mostrado nessa figura, o eixo das ordenadas (y) apresenta o número de ciclos de processamento e o eixo das abscissas (x) apresenta os graus de ofuscação. Cada curva nessa figura representa o número de ciclos de processamento de um programa que foi ofuscado com todas as técnicas de ofuscação de acordo com os graus de ofuscação.

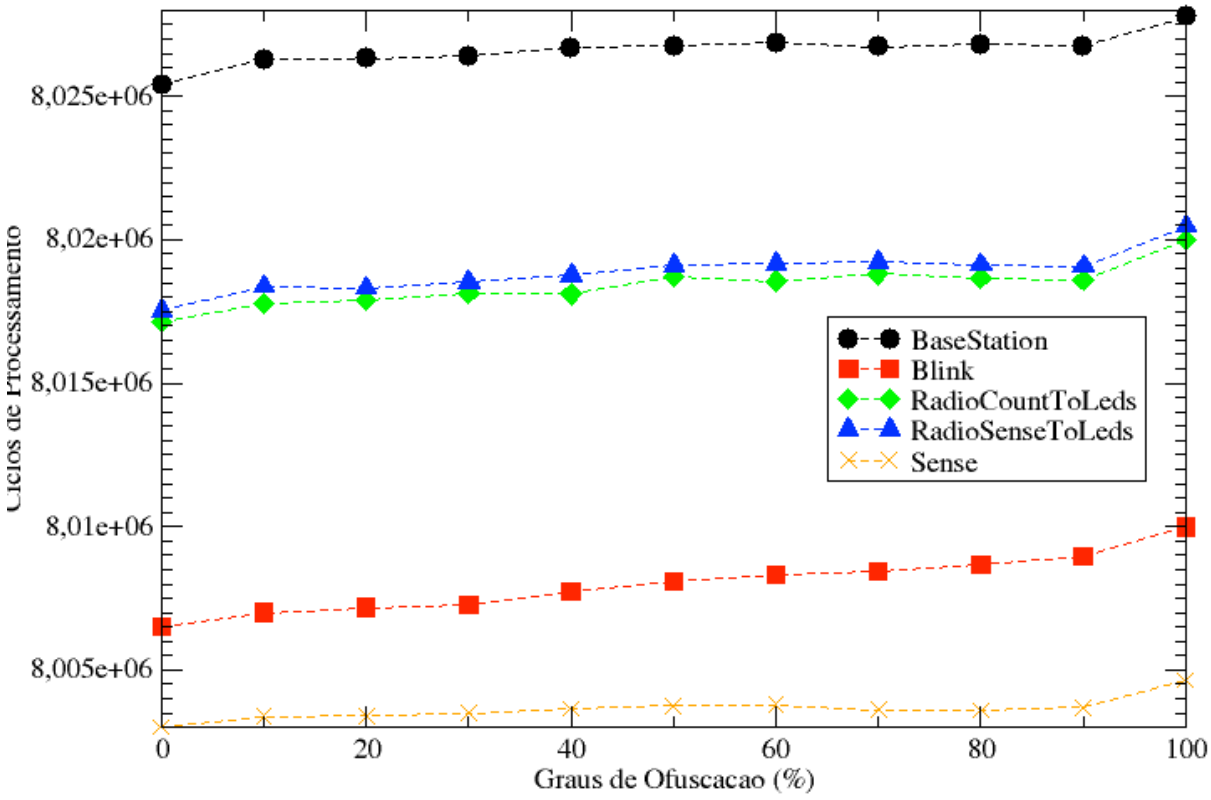

Figura 31. Ciclos de Processamento (número de ciclos).

Cada curva apresentada na Figura 31 mostra que o número de ciclos de processamento dos programas ofuscados aumentaram à medida que o grau de ofuscação foi aumentando. Nesta figura, pode-se notar que à medida que o grau de ofuscação aumenta, ocorre um aumento de aproximadamente 7.9339e-05%, 7.9285e-05%, 1.0497e-04%, 1.7049e-04%, 2.0343e-04%, 1.3470e-04%, 2.0917e-04%, 1.4687e-04%, 1.2924% e 3.2387e-04% no número de ciclos de processamento para respectivamente os seguintes graus de ofuscação: 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% e 100%. De acordo com esses resultados, fica comprovado a afirmação efetuada nesse trabalho que as ofuscações do *TinyObf* não refletem negativamente no aumento dos ciclos de processamento de um programa.

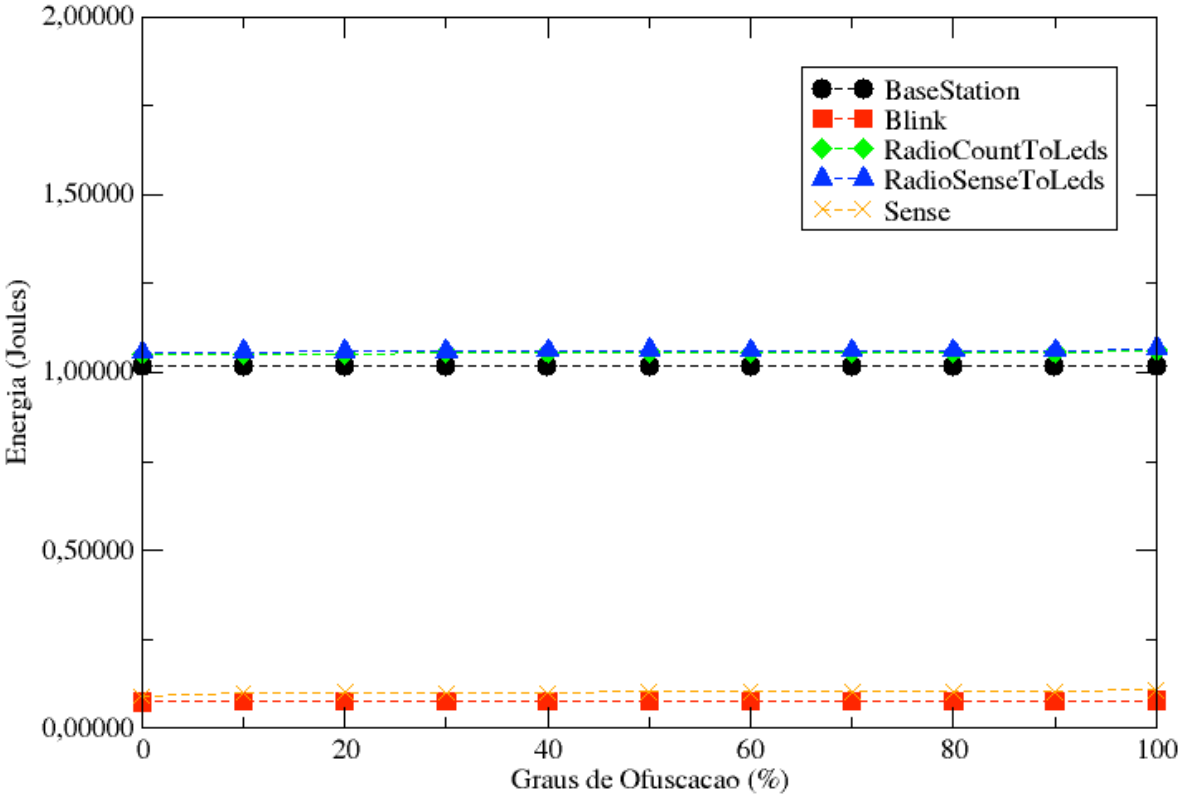

Figura 32. Consumo de Energia (Joule)

A Figura 32 apresenta os resultados obtidos quanto ao consumo de energia dos 5 programas escolhidos para serem ofuscados. Nesse gráfico, o eixo das ordenadas (y) apresenta o consumo de energia em Joules e o eixo das abscissas (x) apresenta os graus de ofuscação. Cada curva nessa figura representa o consumo de energia de um programa que foi ofuscado com todas as técnicas de ofuscação de acordo com os graus de ofuscação. Observe que o grau 0% representa o consumo de energia do programa não ofuscado. Nesta figura é possível notar que à medida que o grau de ofuscação aumenta, não ocorre um aumento significativo no consumo de energia. Em média houve um aumento de aproximadamente 0.02%, 0.0239%, 0.0262%, 0.0275%, 0.0347%, 0.0325%, 0.0346%, 0.0348%, 0.0355%, 0.05% no consumo de energia para respectivamente os seguintes graus de ofuscação: 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% e 100%. De acordo com esses resultados, fica comprovado a afirmação efetuada nesse trabalho que as ofuscações do *TinyObf* não aumentam significantemente o consumo de energia.

1.16 Discussões

Nesta seção, iremos apresentar uma discussão em relação à eficácia da proteção de dados, mostrando argumentos que demonstram que o mecanismo de proteção de dados proposto dificulta a extração de dados confidenciais. Além disso, é apresentada uma outra discussão em relação à resistência do *TinyObf* em relação à desofuscadores automáticos. Isso se deve porque não existem até o presente momento métricas capazes de avaliar tais aspectos.

Eficácia da Proteção de Dados

Medir o esforço necessário para um atacante extrair dados de um programa é muito difícil, pois depende da competência da pessoa que está tentando extrair os dados do programa. No entanto, ao camuflar dados

confidenciais em meio às demais instruções do programa, é possível retardar o atacante, fazendo com que ele seja obrigado a analisar todo o código a fim de identificar quais instruções são na verdade dados confidenciais e, com isso, aumentando a dificuldade de extrair tais dados.

Durante nossos experimentos, utilizamos o *TinyObf* para proteger uma chave criptográfica. Observamos que, em todos os programas analisados, tanto o *avr-objdump* quanto o *IDA PRO* não foram capazes de identificar os bytes dessa chave como dados do programa. Por outro lado, em todos os casos a chave criptográfica *0x0E940102* foi traduzida como se fosse uma chamada de função (CALL *0x0102)*. Dessa forma afirmamos que a proteção de dados proposta nesse trabalho é eficaz, já que os dados confidenciais armazenadas da forma proposta pode ser camuflada em meios às demais instruções do programa.

Resistência à Desofuscadores Automáticos

A resistência à desofuscadores automáticos deve ser avaliada a fim de medir quão difícil é desenvolver um desofuscador capaz de reverter as ofuscações aplicadas em um programa automaticamente. Até o presente momento, não é possível avaliar experimentalmente esse aspecto do arcabouço proposto porque não existem desofuscadores próprios para reverter as ofuscações realizado pelo *TinyObf*. Entretanto, de acordo com Collberg *et al*. (COLLBERG *et al*. 1998), podemos avaliar a resistência à desofuscadores automáticas em termos da dificuldade de desenvolver um desofuscador automático e da dificuldade de um desofuscador reverter a ofuscação. Assim, essa medida pode ser avaliada através do (i) esforço do programador em desenvolver um desofuscador e do (ii) esforço do desofuscador para reverter a ofuscação. O **esforço do programador** é uma medida que calcula o esforço necessário para uma pessoa desenvolver um desofuscador automático capaz de desfazer as ofuscações aplicadas em um programa. Já o **esforço do desofuscador** mede o tempo de execução e a quantidade de memória necessária para o seu funcionamento.

O esforço do programador para desenvolver um desofuscador automático capaz de reverter as ofuscações realizadas pelo *TinyObf* pode ser considerado elevado de acordo com a (i) **dificuldade em identificar se um programa está ofuscado** e a (ii) **dificuldade em identificar quais instruções realizam a ofuscação**. Antes de realizar a desofuscação um desofuscador deve ser capaz de identificar os pontos do programa que estão ofuscados. De acordo com os resultados obtidos na seção 1.15.2, as transformações realizadas pelo *TinyObf* são furtivas em relação à ocorrência de instruções e com isso um programa ofuscado por ele é bastante semelhante a um típico programa não ofuscado. Portanto é difícil desenvolver um desofuscador automático para o *TinyObf*, pois é difícil identificar quais pontos foram ofuscados já que um programa ofuscado com esse arcabouço é muito semelhando com um programa não ofuscado. Uma vez que esse problema foi solucionado, um desofuscador deve ser capaz de identificar quais instruções são necessárias para reverter a ofuscação. No entanto, é difícil identificar quais instruções aplicam uma determinada técnica de ofuscação, pois cada técnica empregada pelo *TinyObf* pode ser realizada através da combinação de diferentes instruções capazes de manter a mesma semântica do programa. Além disso, mesmo que sejam conhecidas todas as combinações associadas a cada técnica de ofuscação, é difícil identificar onde cada uma dessas instruções está localizada já que elas podem estar misturadas com as demais instruções do programa. Com isso a criação de um desofuscador capaz de reconhecer no código quais instruções realizam uma determinada técnica de ofuscação e como substituir essas instruções a fim de desofuscar o programa é uma tarefa muito difícil.

O esforço do desofuscador, por outro lado, pode ser considerado baixo. Uma vez que este é capaz de

identificar as instruções que ofuscam o programa, basta ao desofuscador substituir essas instruções a fim de reverter a ofuscação. Tal substituição não requer muito processamento visto que é somente a modificação de bytes em um arquivo.

Conclusões

Foi apresentado nesse trabalho uma proposta de arquitetura de um arcabouço de ofuscação de código e proteção de dados capaz de melhorar a segurança de uma RSSF. Essa proposta se baseia no uso de técnicas de ofuscação capazes de comprometer as ferramentas de engenharia reversa com o intuito de dificultar o comprometimento dos nós dessa rede. A arquitetura proposta pode ser personalizada para qualquer arquitetura de hardware, desde que as técnicas de ofuscação sejam personalizadas adequadamente. Em nosso estudo de caso, *TinyObf*, nós personalizamos esse arcabouço para uma plataforma de hardware utilizada em RSSFs (*MICAz*) e a partir de nossos resultados esse arcabouço foi considerado uma solução de segurança adequada para RSSFs por conta da eficiência em relação ao consumo de recursos (fator limitante em RSSFs) e dos benefícios trazidos (eficácia da ofuscação e da proteção de dados).

As técnicas de ofuscação empregadas nesse arcabouço são baseadas na substituição e inserção de instruções. A utilização de tais técnicas não impacta de forma negativa nos recursos dos sensores de uma RSSF porque substituem ou inserem poucas instruções ao programa ofuscado. Além disso, essas técnicas são capazes de comprometer tanto a fase de desmontagem como a de descompilação da engenharia reversa, pois induz a tradução incorreta das instruções e a geração errônea de estruturas de alto nível (*ICFG*, *CG* e *CFGs*) a partir do código executável de um programa.

Através dos resultados obtidos em relação ao fator de confusão, seção 1.15.1, verificou-se que o arcabouço proposto pode ser considerado eficaz em termos de dificultar a engenharia reversa, pois à medida que o grau de ofuscação foi aumentando, o fator de confusão também foi aumentando, indicando assim o aumento na dificuldade em realizar a engenharia reversa. Por outro lado, a sobrecarga gerada através do aumento do grau de ofuscação em termos do tamanho do programa, número de ciclos de processamento e consumo de energia não impactam de forma negativa no sensor como visto na seção 1.15.3.

Outro benefício desse arcabouço é que ele pode ser considerado furtivo e resistente à desofuscadores automáticos. Esse arcabouço pode ser considerado furtivo por natureza porque as transformações realizadas pelas técnicas de ofuscação aplicadas por esse arcabouço não dão indícios de que um programa está ofuscado. De acordo com os resultados obtidos na seção 1.15.2 observa-se que um programa ofuscado com todas as técnicas de ofuscação e com 100% de grau de ofuscação continua parecido com um programa não ofuscado. Tal fato é benéfico porque dificulta o desenvolvimento de desofuscadores automáticos. A resistência à

desofuscadores automáticos desse arcabouço também foi discutida e de acordo com os argumentos apresentados (seção 1.16.2), o TinyObf aumenta o esforço para desenvolver um desofuscador capaz de reverte as ofuscações aplicadas por esse acabouço.

A proteção de dados proposta nesse arcabouço mostrou que armazenar dados confidenciais em trechos de código não executáveis pode ser eficaz e eficiente. Eficaz porque os dados confidenciais ficam camuflados em meio às instruções do programa dificultando a sua descoberta e eficiente porque ao adicionar os dados no segmento de código do programa não causa sobrecarga ao dispositivo visto que o tamanho do programa não é alterado, já que os dados somente são realocados do segmento de dados para o segmento de código, sem aumentar o número de ciclos de processamento e nem o consumo de energia visto que esses dados serão acessados da mesma forma que seriam antes de terem sido realocados.

As principais contribuições desse trabalho foram: (i) a descrição de uma arquitetura lógica de um arcabouço de ofuscação de código e proteção de dados eficaz, eficiente e resistente à desofuscadores para RSSF*s* que pode ser personalizado para qualquer arquitetura de hardware; (ii) mecanismo de proteção de dados capaz de camuflar dados confidenciais ao longo do segmento de código de um programa; e (iii) personalização de técnicas de ofuscação existentes na literatura para a arquitetura de processadores *Atmel AVR,* capazes de aumentar a dificuldade de realizar a engenharia reversa, causando pouco impacto sobre os recursos dos sensores. Resultados preliminares relacionados a ofuscação de código e proteção de dados foram publicados na *International Conference on Wireless Networks* (ICWN) (COSTA *et al.* 2012).

1.17 Trabalhos Futuros

Pretendemos aplicar as técnicas de ofuscação diretamente em um código executável de um programa. Assim, o arcabouço de ofuscação de código e proteção de dados se tornaria independente do código fonte. Com isso até mesmo um programa cujo código fonte não esteja disponível poderia se beneficiar do uso desse arcabouço.

Pretendemos avaliar a eficácia desse arcabouço através de outras métricas, como por exemplo, métricas da engenharia de software que avaliam a complexidade de um programa. Outro ponto interessante a ser investigado seria verificar o impacto de inserir outras técnicas de ofuscação à esse arcabouço a fim de torná-lo ainda mais eficaz e resistente à desofuscadores, pois à medida que são utilizadas mais técnicas de ofuscações, o programa ofuscado torna-se mais complexo dificultando a identificação das técnicas de ofuscação utilizadas no programa e aumentando a dificuldade em analisá-lo.

Esperamos ainda refinar esse arcabouço a fim de torná-lo adaptativo. A ideia é aplicar as técnicas de ofuscação do *TinyObf* de forma adaptativa em um ambiente dinâmico. Para isso será necessário desenvolver um mecanismo capaz de tornar o programa ofuscado auto modificável a fim de permitir com que ele mesmo seja capaz de ofuscar e desofuscar trechos de seu código e realocar seus dados confidenciais em tempo de execução de acordo com fatores do ambiente ou do próprio dispositivo. O mecanismo que realizará a ofuscação aumentará a segurança de um programa porque além de comprometer a desmontagem do programa, todo esforço de um atacante durante a engenharia reversa de um programa não será aproveitado ao analisar outro sensor da mesma RSSF devido as distintas versões de código (um para cada sensor). A ofuscação adaptativa também pode proporcionar um maior grau de proteção para os dados confidenciais do programa porque continuamente os trechos de código não executáveis estarão sendo alterados e com isso os dados confidenciais estarão sendo modificados de lugar em tempo de execução.

Referências

ALARIFI, A. ; DU, W. Diversify sensor nodes to improve resilience against node compromise. In: ACM WORKSHOP ON SECURITY OF AD-HOC AND SENSOR NETWORKS, 4., 2006. Alexandria, VA. **Proceedings …** New York: ACM, 2006. p. 101-112.

ALBEROLA, R. ; PESCH, D. AvroraZ: extending avrora with an IEEE 802.15.4 compliant radio chip model. In: ACM PERFORMANCE MONITORING AND MEASUREMENT OF HETEROGENEOUS WIRELESS AND WIRED NETWORKS, 3., 2008, Vancouver. **Proceedings …** New York: ACM, 2008. p. 43-50

ATMEL **Atmel AVR 8-and32-bit microcontrollers.** 2012. Disponível em: http://www.atmel.com/products/microcontrollers/avr/default.aspx. Acesso em:

BALAKRISHNAN, A. ; SCHULZE, C. **Code obfuscation literature survey**. 2005. Disponível em: http://pages.cs.wisc.edu/~arinib/writeup.pdf. Acesso em:

BOCCARDO, D. ; MACHADO, R. ; CARMO, L. Transformações de código para proteção de software, In: SIMPÓSIO BRASILEIRO DE SEGURANÇA DA INFORMAÇÃO E DE SISTEMAS COMPUTACIONAIS, 10., 2010, Fortaleza. **Minicursos …** Fortaleza: SBC, 2010.

CHO, W. ; LEE, I. ; PARK, S. Against intelligent tampering: software tamper resistance by extended control flow obfuscation. WORLD MULTICONFERENCE ON SYSTEMS, CYBERNETICS, AND INFORMATICS, 5., 2001. Orlando. **Proceedings …** [s..l.]: IIIS, 2001

CIFUENTES, C. **Reverse compilation techniques.** 1994. Thesis (Doctor of Philosophy) – , School of Computer Science, Queensland University of Technology, Queensland, 1994.

COLLBERG, C. ; NAGRA, J. **Surreptitious software**: obfuscation, watermarking and tamperproofing for software protection. Upper Saddle River: Addison-Wesley, 2009.

COLLBERG, C. ; THOMBORSON, C. ; LOW, D. **A Taxonomy of obfuscating transformations**. Auckland: Department of Computer Science, The University of Auckland, 1997. (Technical Report, 148).

_____. Manufacturing cheap, resilient, and stealthy opaque constructs. In: ACM-SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 25., San Diego. **Proceedings …** New York: ACM, 1998, p. 184-196.

COSTA, R. et al. TinyObf: Code obfuscation framework for wireless sensor networks, INTERNATIONAL CONFERENCE ON WIRELESS NETWORKS, 11., 2012, Las Vegas. **Proceedings … [**s.l.]: CSREA, 2012.

DAI, C. et al. Static analysis of the disassembly against malicious code obfuscated with conditional jumps. In: INTERNATIONAL CONFERENCE ON COMPUTER AND INFORMATION SCIENCE, 2008, Singapore. **Proceedings ....** Los Alamitos: IEEE, 2008. p.525-530.

DELICATO, F. **Middleware orientado a serviços para redes de sensores sem fio.** 2005. Tese (Doutorado em Ciências em Engenharia Elétrica) – Coordenação dos Programas imbra de Pós-Graduação de Engenharia (COPPE), Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2005.

DI PIETRO, R. et al.. Catch me (if you can) data survival in unattended sensor networks. In: ANNUAL IEEE INTERNATIONAL CONFERENCE ON PERVASIVE COMPUTING AND COMMUNICATIONS, 6., 2008, Hong Kong. **Proceedings …** Los Alamitos: IEEE, 2008. p. 185-194.

GU, Q. Efficient code diversification for network reprogramming in sensor networks. In: ACM CONFERENCE ON WIRELESS NETWORK SECURITY, 3., 2010, Hoboken, NJ. **Proceedings …** New York: ACM, 2010. p. 145-150.

HEX-RAYS. **Executive summary**: IDA PRO - at the cornerstone of IT security. 2012. Disponível em: http://www.hex-rays.com/products/ida/ida-executive.pdf. Acesso em: junho.

HARTUNG, C. ; BALASALLE, J. ; HAN, R. **Node compromise in sensor networks**: the need for secure systems. Boulder: University of Colorado at Boulder, 2005. (Technical Report CU-CS-990-05).

JEONG, G. et al. Generic unpacking using entropy analysis. In: MALWARE 2010 - INTERNATIONAL CONFERENCE ON MALICIOUS AND UNWANTED SOFTWARE, 5., 2010. Nancy. **Proceedings …** [s.l.]: IEEE, 2010. p.98-105.

KRUEGEL, C. et al. Static disassembly of obfuscated binaries. In: USENIX SECURITY SYMPOSIUM, 13., 2004. San Diego. **Proceedings …** Berkeley: USENIX, 2004. p. 255-270.

LAKHOTIA, A. ; KUMAR, E. Abstract stack graph to detect obfuscated calls in binaries. In: IEEE INTERNATIONAL WORKSHOP ON SOURCE CODE ANALYSIS AND MANIPULATION, 4., 2004, Chicago. **Proceedings …** Los Alamitos, IEEE, p. 17-26.

LAKHOTIA, A. et al. Context-sensitive analysis without calling-context. **Journal of Higher-Order and Symbolic Computation**, New York, v. 23, n. 3, p. 275-313, Sept. 2010.

LANDSIEDEL, O. et al. Enabling detailed modeling and analysis of sensor networks. **PIK Journal,** Berlin, v. 28, n. 2, p. 101-106, Jun. 2005. Special Issue on Sensor Networks

LEE, B. ; KIM, Y. ; KIM, J. binOb+: a framework for potent and stealthy binary obfuscation. In: SYMPOSIUM ON INFORMATION, COMPUTER AND COMMUNICATIONS SECURITY, 5., 2010, Beijing. **Proceedings …** New York: ACM, 2010. p. 271-281.

LINN, C. ; DEBRAY, S. Obfuscation of executable code to improve resistance to static disassembly. In: ACM CONFERENCE ON COMPUTER AND COMMUNICATION SECURITY, 10., 2003.Washington, DC. **Proceedings …** New York: ACM 2003. p. 290-299

OGISO, T. et al. Software obfuscation on a theoretical basis and its implementation. **IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science**, Tokyo, v. E86-A, n. 1, .p. 176-186,

2003.

O'SULLIVAN, P. et al. Retrofitting security in COTS software with binary rewriting. In: INTERNATIONAL INFORMATION SECURITY CONFERENCE, 5., 2011, Brno. **Proceedings …** Brno: Brno University, 2011. p. 154-172.

PARK, T. ; SHIN, K. G., Soft tamper-proofing via program integrity verification in wireless sensor networks. **IEEE Transactions on Mobile Computing**, New York, v. 4, n. 3, p. 297-309., May/June 2005.

PELISSIER, S. et al. Providing security in energy harvesting sensor networks. In: IEEE CONSUMER COMMUNICATIONS AND NETWORKING CONFERENCE, 2011. Las Vegas. **Proceedings …** Los Alamitos: IEEE, 2011. p.452-456.

POPOV, I. ; DEBRAY S. ; ANDREWS G. Binary obfuscation using signals. USENIX SECURITY SYMPOSIUM, 16., 2007, Boston. **Proceedings …** Berkeley USENIX, 2007. p. 275-290.

SAVANNAH. **AVR C Runtime Library**. 2012. Disponível em: http://savannah.nongnu.org/projects/avr-libc/. Acesso em: jun.

SCHWARZ, B. ; DEBRAY, S. ; ANDREWS, G. Disassembly of executable code revisited. In: IEEE WORKING CONFERENCE ON REVERSE ENGINEERING, 9., 2002, Richmond. **Proceedings …** Los Alamitos: IEEE, 2002. p. 45-54

SHAMIR, A. ; SOMEREN, N. (1999) Playing Hide and Seek with Stored Keys. In: INTERNATIONAL CONFERENCE ON FINANCIAL CRYPTOGRAPHY, 2002, Anguilla, British West Indies. **Proceedings …** London: Springer-Verlag, 1999. p. 118-124.

SHARIR, M. ; PNUELI, A. Two approaches to interprocedural data flow analysis. In: JONES, N. D. ; MUCHNICK, S. S. **Program flow analysis**: theory and applications. [s.l.]: Prentice-Hall, 1981. p. 189-234. (Prentice-Hall Software Series)

SMITHSON, M. et al. **Binary rewriting without relocation information**. College Park: University of Maryland, 2010. (Technical Report)

TINYOS. **TinyOS is an open source, BSD-licensed operating system designed for low power wireless devices.** 2012. Disponível em: http://www.tinyos.net. Acesso em jun.

VIEGA, J. ; MESSIER, M. **Secure programming cookbook for C and C++**. Beijing: O'Reilly Media Publisher, 2003.

WANG, C. et al. **Software tamper resistance**: obstructing static analysis of programs. Charlottesville: Dept. of Computer Science, University of Virginia. 2002. (Technical Report, CS-2000-12).

WANG, Y. ; ATTEBURY, G. ; RAMAMURTHY, B. A survey of security issues in wireless sensor networks. **IEEE Communications Surveys & Tutorials**. New York: v. 8, n. 2, p.2-23, 2^0 Quarter 2006.

WARTELL, R. et al. Differentiating code from data in x86 binaries. In: EUROPEAN CONFERENCE ON MACHINE LEARNING AND KNOWLEDGE DISCOVERY IN DATABASES, 2011, Athens. **Proceedings …** Berlin: Springer-Verlag, 2011.

YICK, J.; MUKHERJEE, B. ; GHOSAL, D. Wireless sensor network survey. **Computer Networks**, Amsterdam, v. 52, n. 12, p. 2292-2330, Aug. 2008.

ZIA, T. ; ZOMAYA, A. Security issues in wireless sensor networks. In: INTERNATIONAL CONFERENCE ON SYSTEMS AND NETWORKS COMMUNICATION 2006, Tahiti. Proceedings … Piscataway: IEEE, 2006. p.40.