Defending Internet of Things against Exploits

F. A. Teixeira, G. V. Machado, P. M. Fonseca, F. M. Q. Pereira, H. C. Wong, J. M. S. Nogueira, L. B. Oliveira

Abstract— **The Internet of Things (IoT) demands tailor-made security solutions. Today, there are a number of proposals able to meet IoT's demands in the context of attacks from outsiders. In the context of insiders, however, this does not hold true. Existing solutions to deal with this class of attacks not always take into consideration the IoT's idiosyncrasies and, therefore, they do not produce the best results. This work aims at coming up with tailor-made security schemes for thwarting attacks from insiders in the context of IoT systems. Our solution makes use of a pioneering solution to pinpoint vulnerabilities: we crosscheck data from communicating nodes. It provides the same guarantees as traditional security mechanisms, but it is about 83% more efficient, according to the experiments that this article describes.**

Keywords— **Code Security, Internet of Things, Buffer Overflow.**

I. INTRODUÇÃO

A INTERNET das Coisas (IoT – *Internet of Things*) é uma infraestrutura de rede dinâmica e global com capacidades de autoconfiguração onde "*coisas*" físicas e virtuais têm atributos físicos, personalidades virtuais e identidades. Tais *"coisas"* usam interfaces inteligentes bem como são naturalmente integradas à Internet [1]. Na IoT, as "*coisas"* ou objetos tornam-se participantes ativos em processos de negócio, informacionais e sociais, onde são capazes de interagir e comunicar entre si, trocar informações coletadas do ambiente, reagir autonomamente aos eventos do mundo físico real, bem como influenciar esse contexto com ou sem intervenção direta do ser humano.

Mas questões como segurança e privacidade devem ser consideradas [27]. Ataques a esse tipo de rede já começam a ser reportados. Como exemplo podemos citar o recente envio de SPAMs através de roteadores domésticos, TVs e até geladeiras que foram usados como "*Thingbots"* [2]. Para proteger a IoT são necessárias muitas camadas de segurança incluindo uma que proteja a IoT de exploração de falhas de código via manipulação das entradas. Entre exemplos de ataques desse tipo em sistemas tradicionais podemos citar o Estouro de Arranjo (*BoF - Buffer Overflow*) [11], Estouro de Inteiro (*IoF – Integer Overflow*) [12] e a Injeção de SQL (*SQL Injection*) [3]. Acreditamos que ataques desse tipo serão frequentes em sistemas da IoT também. Por exemplo, um adversário poderá descobrir combinações de teclas do controle remoto de uma TV que a induz a um Estouro de Arranjo. Ou, poderá acender um isqueiro próximo a um sensor de temperatura para explorar uma falha de Estouro de Inteiro [12] e comprometer uma Rede de Sensores Sem Fio (RSSF) [26].

F. A. Teixeira, UFMG, Belo Horizonte, Brasil, teixeira@dcc.ufmg.br
G. M. Vieira, UFMG, Belo Horizonte, Brasil, gvieira@dcc.ufmg.br
P. M. Fonseca, UFMG, Belo Horizonte, Brasil, pablom@dcc.ufmg.br
F. M. Q. Pereira, UFMG, Belo Horizonte, Brasil, fernando@dcc.ufmg.br
J. M. S. Nogueira, UFMG, Belo Horizonte, Brasil, jmarcos@dcc.ufmg.br
L. B. Oliveira, UFMG, Belo Horizonte, Brasil, leob@dcc.ufmg.br
H. C. Wong, Intel Corporation, Santa Clara, EUA, hao-chi.wong@intel.com

A Internet das Coisas (IoT – Internet of Things) requer soluções de segurança feitas sob medida [25].

Isso porque, ao contrário de um computador tradicional, os elementos de IoT possuem características especificas [1]. Por exemplo, os dispositivos da IoT usualmente: (i) são dotados de menor capacidade de processamento, memória e energia; (ii) realizam tarefas de forma colaborativa; e (iii) executam sistemas eminentemente desenvolvidos usando a linguagem C ou sistemas operacionais baseados em C como o TinyOS [4], ContikiOS [5] e Linux embarcado.

O objetivo deste trabalho é conceber uma solução de segurança de código para IoT capaz de defender seus sistemas contra ataques internos à rede, em especial contra Estouro de Arranjos. Nossa proposta, emprega uma abordagem pioneira na busca de vulnerabilidades: o cruzamento de dados de diferentes módulos de um mesmo sistema distribuído. Esse cruzamento de dados nos permite ter uma visão holística do sistema e maior precisão. Tal precisão é traduzida em eficiência. Até onde sabemos, não há propostas que usem técnicas de cruzamento de dados de diferentes módulos do sistema para este mesmo fim.

As principais contribuições deste trabalho são as seguintes:

- Concepção de um algoritmo para unificar o grafo de fluxo de controle de diferentes programas em um Sistema Distribuído.
- Extensão de duas estruturas de dados usadas na área de compiladores para representar sistemas distribuídos – grafo de fluxo de controle e grafo de dependências.
- Projeto e desenvolvimento de uma ferramenta para analisar o código fonte de diferentes programas de um Sistema Distribuído como uma única aplicação. A análise proposta emprega uma técnica de cruzamento de dados para apontar arranjos sujeitos a ataques de Buffer Overflow. A primeira versão dessa ferramenta foi descrita em [25] e encontra-se disponível online [24].
- Avaliação da solução usando aplicações do *ContikiOS* [5], um Sistema Operacional voltado para IoT.

Acerca do último ponto, para avaliar a solução proposta foram feitos experimentos com quatro aplicações do *Contiki OS*. Os resultados indicam que para um mesmo nível de segurança, a solução proposta se mostrou pelo menos 83% mais eficiente que propostas hoje amplamente utilizadas.

O restante deste artigo está organizado da seguinte maneira. A Seção II. descreve o modelo de ataque e conceitos de segurança de código através de um exemplo. A solução proposta é apresentada na Seção III. A Seção IV. apresenta um estudo de caso e os resultados dos experimentos. Os trabalhos relacionados são discutidos na Seção V. As conclusões e trabalhos futuros são apresentados na Seção VI.

II. MODELO DE ATAQUE

Nessa seção, descrevemos o modelo de ataque considerado nesse artigo e apresentamos um exemplo de análise estática de código onde a visão global do sistema ajuda a reduzir os falsos

positivos, comuns nesse tipo de análise.

A. Premissas & Modelo de Ataque

O modelo de ataque deste trabalho pressupõe que: (i) As mensagens transmitidas pela rede são seguras e os programas de cada vértice são íntegros; (ii) os adversários conhecem o código fonte e tem acesso aos canais de entrada do sistema. A primeira suposição foi tomada para definir o escopo da nossa proposta. Muitos mecanismos de segurança destinam-se a garantir que as mensagens trocadas através da rede são seguras e que os programas instalados estão intactos [26]-[28]. No entanto, mesmo que esses mecanismos obtenham sucesso, não podemos afirmar que o sistema estará seguro. Adversários com acesso ao sistema podem explorar falhas no código e realizar ataques através de seus canais de entrada (suposição ii). A solução proposta nesse artigo visa proteger as aplicações contra esse tipo de ataque.

Por exemplo, um adversário pode estudar o código fonte de programas de código aberto instalados em uma TV em busca de vulnerabilidades. A partir das vulnerabilidades do código o adversário pode manipular as entradas da TV (combinação de teclas do controle remoto, entrada HDMI, etc.) a fim de provocar um Estouro de Arranjo no sistema. Como as "*coisas*" estão ligados à Internet, o objeto comprometido (a televisão nesse caso) pode ser utilizada como um agente para atacar outro dispositivo na rede também.

No contexto de IoT esse tipo de ataque pode tomar grandes proporções uma vez que a conexão entre diversos dispositivos e a Internet pode ser explorada para realizar ataques em cascata onde um dispositivo comprometido replica o ataque para outros dispositivos na rede. A seguir, apresentamos um exemplo para ilustrar como a análise de segurança de código pode ser usada para detectar esse tipo de falha no código mas também como a falta de uma visão global do sistema pode gerar falsos positivos.

B. Exemplo de Análise de Segurança de Código

O gráfico de controle de fluxo (CFG) é uma estrutura de dados central na análise e na otimização de programas. Desde sua introdução, por Frances Allen, no início dos anos setenta [8], praticamente todo compilador atual usa alguma forma de CFG para representar programas. O CFG de um programa P é um grafo dirigido definido da seguinte forma: para cada instrução $i \in P$, cria-se um vértice v_i. Adiciona-se uma aresta de v_i para v_j se é possível executar a instrução j imediatamente após a instrução i. A Figura 1 mostra dois exemplos de CFGs.

Muitas ferramentas de análise estática, como a proposta nesse artigo, utilizam os CFGs e reportam para os desenvolvedores avisos sobre a segurança de trechos de código de uma aplicação. O CFG tradicional, porém, não é capaz de representar as interações entre programas de um sistema distribuído mas apenas cada uma de suas partes. Por isso, algumas das advertências levantadas por ferramentas tradicionais não representam vulnerabilidades reais em um Sistema Distribuído. Advertências desse tipo, são chamadas de falsos positivos. Enquanto vulnerabilidades reais no código, são chamados de verdadeiros positivos.

Por exemplo, na linha 5 do programa servidor da Figura 1b, há um verdadeiro positivo. O arranjo `msg` é carregado com dados que vem da rede através da função `recv`. Esta instância de `recv` lê os dados a partir do segundo `send` do programa cliente - linha 6 da Figura 1a. E este `send`, por sua vez, depende do `getc` (linha 4), uma fonte local de dados. Como existe um caminho para um arranjo a partir de uma fonte local de dados, que o adversário pode manipular, o nosso exemplo contém um verdadeiro positivo.

Entretanto, na mesma figura, temos um exemplo de um falso positivo. Se analisarmos apenas o programa servidor em nosso sistema (Figura 1b), concluímos que o arranjo `msg` é vulnerável, uma vez que depende do primeiro `recv`, na linha 1 da Figura 1b. No entanto, se olharmos o sistema como um todo, esta vulnerabilidade desaparece. A operação `recv` da linha 1 só lê as informações do primeiro `send`. Este envio sempre transmite uma constante para o outro programa. Como o adversário não pode manipular essa informação, esse trecho de código é seguro.

Portanto, a fim de melhorar a precisão da análise de fluxo de dados, precisamos de uma representação holística do sistema, o que pode ser obtida através do que chamamos de Grafo de Fluxo de Controle Distribuído (DCFG – *Distributed Control Flow Graph*) que será descrito na próxima seção.

III. PROPOSTA DE SOLUÇÃO

Entre as estruturas de dados usadas na análise de código, duas são especialmente importantes no contexto deste artigo: o Grafo de Fluxo de Controle (CFG) [8] e o Grafo de Dependência (DG) [9]. A proposta deste artigo estende essas estruturas para representar um sistema distribuído como um todo. Nessa seção apresentamos o *Grafo de Fluxo de Controle Distribuído – DCFG e o Grafo de Dependência Distribuído – DDG*. Em seguida, descrevemos a implementação da solução proposta.

A. Grafo de Controle de Fluxo Distribuído

Apesar da grande importância do *CFG*, neste artigo, defende-se que esta estrutura de dados não é expressiva o suficiente para representar programas que se comunicam através de uma rede. Essa deficiência decorre do fato de que o CFG, em sua concepção original, foi projetado para representar programas individuais e não aplicações distribuídas. Para suprir esta deficiência, propomos um Grafo de Fluxo de Controle Distribuído, ou DCFG. Este grafo une os CFGs dos programas individuais que constituem o sistema distribuído.

O DCFG é a união dos CFGs. São inseridas arestas entre um SEND de um CFG com o RECV de outro. Sua construção pode ser sumarizada nos seguintes passos: (i) extrair grafos de mensagens com SEND/RECV para cada programa; (ii) determinar os níveis de cada SEND/RECV; (iii) ligar SENDs de um programa a RECV de outro que estejam em um mesmo nível. A seguir detalhamos esses passos.

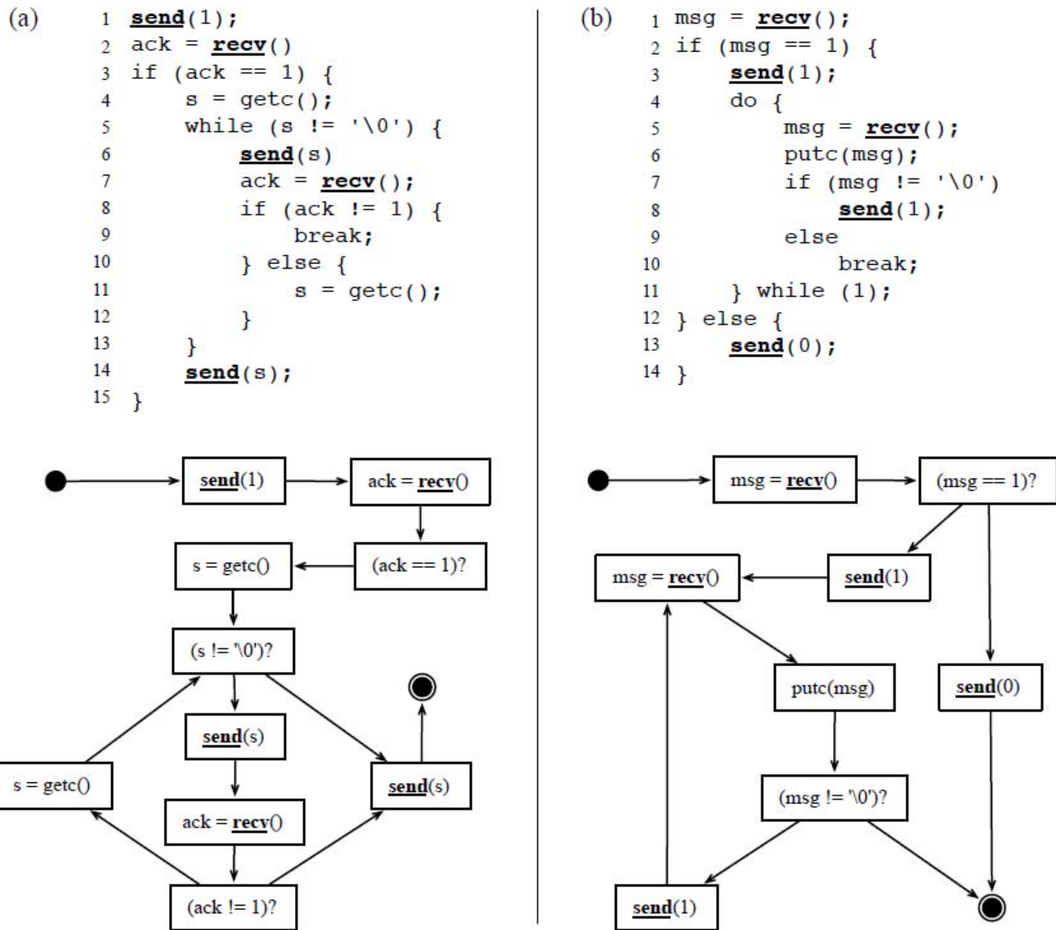

Figura 1. Os grafos de fluxo de controle de dois programas que constituem uma aplicação echo. (a) O programa cliente. (b) O servidor echo.

Para unir os CFGs, começamos por extrair o grafo de funções SEND (*SG*) e os grafo de funções RECV (*RG*), para cada programa. Esses grafos são visões do programa que sintetizam o padrão de comunicação definido por cada *CFG*. Para cada vértice $send \in CFG_i$, adiciona-se um vértice v' ao SG_i. Adiciona-se também *entry*' e *exit*' em cada *SG*. Estes vértices são versões dos nós de *entrada* e *saída* do CFG_i. Arestas no SG_i correspondem a caminhos entre SENDs no CFG original. Ou seja, para cada par de vértices $u, v \in CFG_i$, adiciona-se uma aresta u', v' no SG_i se, e somente se: (i) existe um caminho p de u para v no CFG_i, e (ii) p não contém nenhum outro SEND. O RG_i é criado maneira análoga, substituindo SENDs por RECVs no procedimento descrito acima. O *SG* e *RG* do cliente *Echo* do nosso exemplo podem ser vistos nas Figura 2 e Figura 3 respectivamente.

A partir dos *SG* e *RG* de cada programa construímos o *DCFG.* Seja $\{C_1, C_2, \dots, C_n\}$ o conjunto dos *CFGs* que constituem um sistema, $\{S_1, S_2, \dots, S_n\}$ os grafos *SEND* e $\{R_1, R_2, \dots, R_n\}$ os grafos *RECV* para cada programa. O *DCFG* que contem cada $CGF \in \{C_1, C_2, \dots, C_n\}$ como um subgrafo. Além disso, ele contém uma aresta que liga cada vértice *SEND* e *RECV* que podem se comunicar. Para cada par de nós $s_i \in S_q$ e $r_i \in R_p$, adicionamos uma aresta *entre programas* de s_i e r_j, se existe uma sequência de execução em que uma mensagem enviada por s_i alcança r_j.

Em princípio, podemos adicionar arestas entre programas conectando cada vértice SEND a todo vértice RECV. O *DCFG* resultante é válido se assumirmos que cada par de *SEND/RECV* tem uma relação em potencial. Contudo, isto é conservativo em excesso, dado que muitos nós não podem trocar mensagens. Por exemplo, os nós A e H, na

Figura *4*, não podem se comunicar. Toda mensagem que A envia, será recebida pelo vértice F antes que H tenha a chance de executar.

Para encontrar uma aproximação de DCFG melhor que a solução trivial, nós introduzimos a noção de *níveis de nós* (Equação 1). O Nível 0 contém o vértice raiz. O Nível 1 contém os SENDs alcançados diretamente pela raiz, em outras palavras, contém os sucessores imediatos dos vértices do nível anterior. E assim sucessivamente, o Nível $n + 1$ contém os sucessores dos vértices no Nível n. O algoritmo termina quando o final do grafo é alcançado ou quando um conjunto de níveis é repetido. Apenas elementos no mesmo nível podem se comunicar, porque, intuitivamente, o nível de um vértice determina o número de saltos até aquele nó, iniciando a partir do vértice raiz.

$$
\begin{aligned}
nível(cfg, 0) &= \{raiz\} \\
nível(cfg, n) &= \{v \mid \overrightarrow{uv} \in nível(cfg, n-1)\} \qquad (1)
\end{aligned}
$$

Para ilustrar essa intuição, a Figura 2 mostra os níveis para os SENDs do cliente *Echo* do nosso exemplo. O primeiro,

nível 0, contém a raiz do grafo. O nível 1 contém o sucessor imediato do vértice raiz: $\{A\}$. O nível 2 contém os sucessores imediatos de cada vértice SEND do nível 1: $\{C, E\}$. Ao computar o nível 3, verifica-se que os sucessores de *C* são $\{C, E\}$ e não há sucessores de E, logo o nível 3 é igual ao nível 2. Portanto, o algoritmo termina. De maneira similar, encontramos os níveis do grafo de *RECV* do servidor *Echo* (Figura 3). Apenas o vértice $\{F\}$ é alcançável diretamente a partir da raiz, portanto constitui o nível 1. O nível 2 é composto por $\{H\}$, sucessor imediato do vértice F. Os níveis seguintes também são compostos por $\{H\}$, visto que há um laço nesse vértice que só termina como fim do programa. A

Figura *4* mostra as ligações entre os SENDs do cliente e os RECVs do servidor do nosso exemplo. Os números entre chaves são os níveis previamente computados. Apenas nós de mesmo nível são interligados.

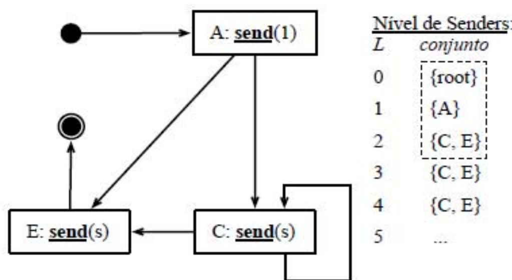

Figura 2. Níveis dos SENDs do servidor *Echo*.

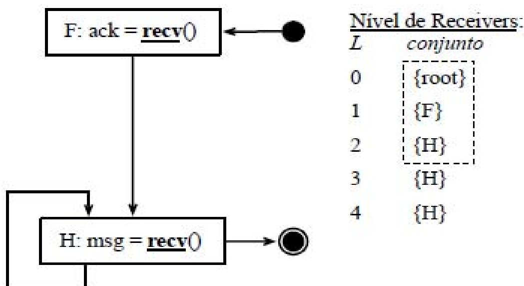

Figura 3. Níveis de RECVs do cliente *Echo*.

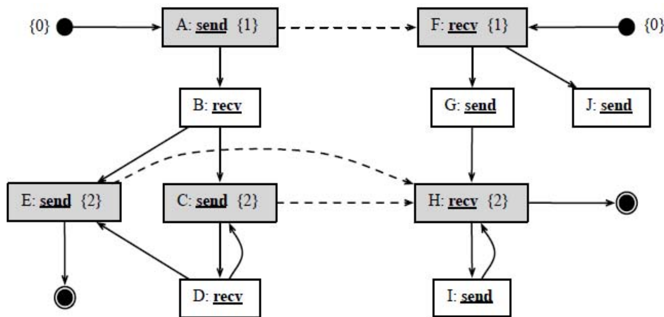

Figura 4. Linhas tracejadas representadas as ligações entre o cliente e o servidor *Echo* do nosso exemplo.

B. *Grafo de Dependência Distribuído*

Enquanto o CFG representa o fluxo de controle de um programa, os Grafos de Dependência (DG) [9] foca no fluxo de dados, i.e., dependências entre as instruções e os dados. O DG contém um vértice para cada variável e operação no programa. Existe uma aresta de uma variável *v* para uma operação *i* se *i* for uma instrução que usa *v*. Da mesma forma, esse grafo contém uma aresta de *i* até *v* se *i* for uma variável *v*.

A construção do DDG (*Grafo de Dependência Distribuída*) segue uma metodologia similar usada na construção do DCFG. Como o DDG é a união dos grafos de dependência de programas individuais, inicialmente, são criados os grafos de dependência de cada programa do sistema distribuído. Em seguida, para cada instrução que acessa a rede, adiciona-se um vértice no grafo para representar essa operação. Finalmente, os níveis definidos nos grafos de SEND e RECV são usados para decidir que arestas devem ser inseridas entre os SENDs e *RECVs*, de maneira similar à descrita anteriormente. O grafo final contém as dependências de todos os programas do sistema distribuído como se fosse um único programa.

O DDG pode ser usado para diversos tipos de análise de segurança como *buffer overflow* ou *integer overflow*. Por exemplo, na Seção IV. descrevemos como utilizamos o DDG em nosso estudo de caso para encontrar dependências entre entradas de usuários e arranjos que podem ser utilizado para ataques de *buffer overflow* e, ao mesmo tempo, mitigar falso positivos devido ao acesso de rede.

C. *Arquitetura e Implementação*

A solução proposta foi implementada como uma extensão do compilador LLVM [10]. A Figura 5 mostra a arquitetura da solução proposta. O programa é analisado em três camadas: (i) Entrada, (ii) Núcleo, e (iii) Estudo de Caso. Incialmente, cada programa é convertido na representação intermediária (IR) do LLVM como um arquivo de *bytecode*. Na camada de Entrada, produzimos um bloco de código único a partir da união de cada programa, chamamos esse estágio de M*erge*. Em seguida, na camada Núcleo, através do passo Elevador extraímos os grafos de SENDs e RECVs de cada programa, definimos o nível e produzimos o *DCFG*. A partir do *DCFG* e do *bytecode* unificado nós produzimos o *DDG* com o DistDepGraph. Finalmente, como estudo de caso, analisamos as dependências entre entradas e memórias para encontrar arranjos vulneráveis. Damos como saída os grafo dos caminhos que mostram arranjos vulneráveis, assim como dados estatísticos do programa.

No primeiro estágio que descrevemos no parágrafo anterior, a produção dos arquivos de *bytecode*, requer intervenção do usuário, pois precisamos determinar que funções acessam a rede. Nossa ferramenta analisa todas as funções de rede apresentadas em cada arquivo de *bytecode*, baseada em bibliotecas usuais da linguagem C. Então damos como saída para o usuário essa lista, que rege nossos candidatos em potencial. O usuário determina que função transmitem, e que funções recebem dados. A partir dessa informação adicionamos anotações aos arquivos produzidos pelo LLVM, identificando SENDs/RECVs. Os arquivos anotados são então mesclados em um único *bytecode* para facilitar análises subsequentes.

Na camada Núcleo, após o estágio de unificação, determinamos que funções podem ler dados de entradas não confiáveis. Por entrada não confiável nós denotamos qualquer função que possa interagir com o usuário ou com o ambiente. Em seguida, o *Elevador*, implementa o algoritmo que descobre níveis, que nós descrevemos na seção anterior e cria o DCFG. Baseado no DCFG, o *DistDepGraph*, constrói o DDG.

Finalmente, na camada de Estudo de Caso, nós passamos o DDG para *DistVulArrays*, o quarto e último passo em nosso fluxo de análise. Esse passo procura por caminhos entre pontos não confiáveis e operações sensíveis. Após analisar um programa, DistVulArrays produz as seguintes saídas: (i) o número de verdadeiro-positivos, e potenciais falso-positivos; e (ii) o grafo dos caminhos vulneráveis em cada programa.

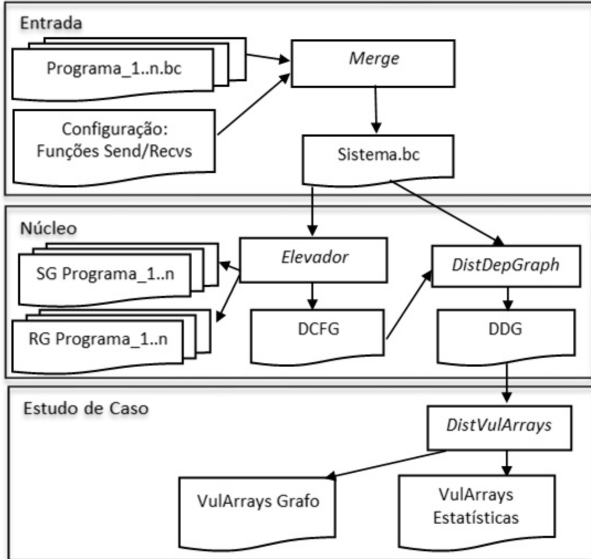

Figura 5. Arquitetura.

IV. Estudo de Caso e Resultados

Ataques de BoFs (*Buffer Overflows*) são um dos tipos mais usados para obter o controle de sistemas [11]. Eles são evitados por *array-bounds checks* - ABCs [8]. Ou seja, o código de instrumentação verifica se um índice está dentro dos limites de um arranjo antes de acessar a memória. Contudo, o uso de ABCs consome recursos de energia e memória de forma que se forem usados de maneira indiscriminada sua aplicação torna-se inviável. As soluções existentes são conservadores e produzem um número considerável de falso-positivos, ou seja, muitos ABCs podem ser eliminados. Nesta seção, descrevemos como a solução proposta foi utilizada para proteger sistemas de IoT contra BOFs, inserindo ABCs apenas em pontos que possuem ligação com entradas de usuários. Dessa forma, foi possível reduzir o número de ABCs necessários para proteger uma aplicação de IoT. A proposta foi avaliada utilizando as seguintes aplicações do ContikiOS: (i) udp-ipv6 cliente, (ii) udp-ipv6 servidor, (iii) webserver e (iv) wget.

As aplicações udp-ipv6 (https://github.com/contiki-os/contiki/tree/master/examples/udp-ipv6) demonstram como fazer um cliente e servidor baseado em UDP 6LoWPAN (IPv6 sobre Low power Wireless Personal Area Networks – http://tools.ietf.org/html/rfc6282) . No servidor UDP (Figura 6) as mensagens são recebidas pela rede pela função `uip_newdata` (linha 4), e são acessadas pelo arranjo `uip_appdata`. As ferramentas tradicionais consideram esse arranjo vulnerável e introduzem um ABC para protegê-lo. O ABC é acionado assim que uma mensagem é recebida. Além disso, no módulo cliente (Figura 7) o outro arranjo seria protegido por ferramentas tradicionais. Pois a função `uip_newdata` (linha 3) é chamada para verificar mensagens recebidas e então os dados são transferidos para o arranjo `str` (linha 4).

```
 1  static void tcpip_handler(void) {
 2    static int seq_id;
 3    char buf[MAX_PAYLOAD_LEN];
 4    if(uip_newdata()) {
 5      ((char *)uip_appdata)[uip_datalen()] = 0;
 6      PRINTF("Server received: '%s' from ", (char
             *)uip_appdata);
 7      ...
 8      uip_ipaddr_copy(&server_conn->ripaddr,
    &UIP_IP_BUF->srcipaddr);
 9      PRINTF("Responding with message: ");
10      sprintf(buf, "Hello from the server! (%d)",
            ++seq_id);
11      PRINTF("%s\n", buf);
12      uip_udp_packet_send(server_conn, buf,
                          strlen(buf));
13      ...
14    }
15  }
```

Figura 6. Trecho do Código do Servidor udp-ipv6 do Contiki Os.

```
 1  static void tcpip_handler(void) {
 2     char *str;
 3     if(uip_newdata()) {
 4     str = uip_appdata;
 5     str[uip_datalen()] = '\0';
 6     printf("Response from the server: '%s'\n", str);
 7    }
 8  }
 9  static void timeout_handler(void) {
10    static int seq_id;
11    printf("Client sending to: ");
12    PRINT6ADDR(&client_conn->ripaddr);
13    sprintf(buf, "Hello %d from the client",++seq_id);
14    printf(" (msg: %s)\n", buf);
15    uip_udp_packet_send(client_conn,buf,UIP_APPDATA_SIZE);
16  ...
17  }
```

Figura 7. Trecho do Código do Cliente udp-ipv6 do Contiki OS.

Mas, quando analisamos o sistema como um todo, é possível observar que os arranjos acima não são vulneráveis. Observe que a mensagem preparada pelo cliente não tem nenhuma dependência com fontes de dados externas (Figura 7, linhas 10-14). Se esta mensagem não é vulnerável em sua origem, seguindo nosso modelo de ataque (Seção II. A.), ela também não é vulnerável em seu destino. No servidor, da mesma forma, a mensagem enviada pelo cliente não depende de dados de entrada (Figura 6, linhas 8-11). Então, não é necessário inserir ABCs nesse caso.

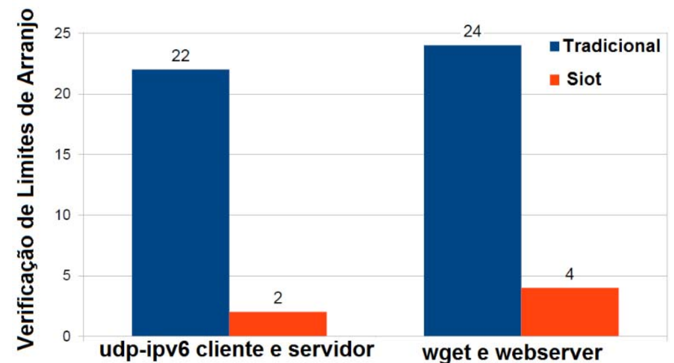

Figura 8. Número de Verificações de Limites de Arranjos .

O número de ABCs necessários para proteger as aplicações *udp-ipv6*, assim como as aplicações *webserver* e *wget* pode ser visto na

Figura 8. Por exemplo, quando analisamos aplicações udp-ipv6 como um todo, a abordagem tradicional aponta 24 arranjos que precisam de verificação de limites enquanto nossa proposta aponta apenas 4. A diferença, de 20 ABCs acontece porque a abordagem tradicional considera vulneráveis todos os dados recebidos pela rede, enquanto nossa abordagem utiliza a verificação cruzada de dados para confirmar se eles são vulneráveis ou não. Dessa forma, podemos reduzir em 91% o número de ABCs inseridos na aplicação udp-ipv6 (22 para 2) e 83% nas aplicações webserver/wget (24 para 4).

O número de ABCs que podemos eliminar é importante principalmente em termos da energia poupada, como podemos ver na Tabela I. Para cada ABC é necessário incluir 6 instruções assembly [9]. Se considerarmos um dispositivo de IoT típico com um micro controlador Atmel AVR de 8 mA, 3v, 7,38MHz e 1 instrução/ciclo, o consumo energético será de 3,25 mJ/instrução (Energia=I*V*tempo onde I é a corrente, V a tensão e tempo=instruções/ciclo.). Então, cada novo ABC consome 19,5 mJ. Se uma aplicação upd-ipv6 receber uma mensagem por minuto, então ela irá consumir 618 mJ na abordagem tradicional contra 56 mJ com a nossa proposta. Podemos ver na Tabela I o impacto dessa economia na prática. Em uma rede com 100 dispositivos executando essa aplicação durante uma semana, a economia corresponderá a quinta parte da capacidade de uma bateria alcalina (O preço da bateria alcalina de R$4,40 e a energia por bateria 3*Wh*.), ou seja, 0,04 kJ ou R$0,84. No final do ano, a economia alcança 1,86 baterias ou R$10,21.

TABELA I. ECONOMIA DE ENERGIA.

	Economia de Energia (kJ)	Pilhas (AA) Economizadas	Economia Monetária (R$)
Uma Semana	0,393	0,04	0,10
Um Mês	1,685	0,16	0,40
Um Ano	20,223	1,86	4,64

TABELA II. TAMANHO DAS APLICAÇÕES ANALISADAS.

Aplicação	Instruções	DCFG Vértices	DCFG Arestas	Tempo (s)	Memória (MB)
udp-ipv6 Cliente Servidor	42.904	65.218	95.237	33.25	45.40
Webserver e wget	42.828	66.154	93.678	27.28	92.53

Finalmente, é importante notar que a análise estática que propomos não requer recursos computacionais excessivos. Para executá-la gastou-se uma média de 29,5 segundos em um notebook com processador de 2.20GHz Intel core i7 (Tabela II). Cada aplicação analisada tem mais de 49.000 instruções e cada programa contém cerca de 65.000 vértices e 90.000 arestas. A análise consumiu 45,4 MB de memória *RAM* para udp-ipv6 e 92,53 MB para a aplicação wget / webserver.

V. TRABALHOS RELACIONADOS

Boa parte das propostas para segurança de código são baseadas em Análise Estática, Análise Dinâmica ou uma combinação das duas [9],[13],[14]. O desenho dessas propostas, no entanto, é uma tarefa desafiadora já que qualquer propriedade não trivial de linguagens recursivamente enumeráveis é um problema não decidível. Em outras palavras, não há um programa genérico capaz de decidir de um programa é vulnerável ou não. Na Análise Estática [8], [15] o sistema é inspecionado antes da execução do programa. Por esta razão, a Análise Estática é também conhecida como *análise de código*. A vantagem desse método é que não há sobrecarga sobre o sistema em execução. A sua desvantagem é que essa análise não tem algumas informações que só estão disponíveis durante a execução. Não tão raro essa "desinformação" impossibilita dizer quando há uma vulnerabilidade em certas partes de um sistema. Na dúvida, a análise é conservadora: assume que a vulnerabilidade existe. O conservadorismo se traduz em falso-positivos.

Na Análise Dinâmica [9], o sistema é analisado durante sua execução. Utilizando a vantagem das informações disponíveis apenas em tempo de execução, esta técnica mitiga o problema dos falso-positivos, comuns na Análise Estática. Porém, seus resultados são relevantes apenas para as entradas testadas e portanto não podem trazer conclusões sobre o comportamento geral do programa. Devido à natureza complementar das análises Estática e Dinâmica, é comum o uso de uma Análise Híbrida, uma combinação das duas [13]. Normalmente, a Análise Estática é usada primeiramente para apontar as vulnerabilidades e em seguida a instrumentação da Análise Dinâmica monitora o sistema em tempo de execução, nos pontos supostamente vulneráveis. Mesmo que eles não apresentem nenhum perigo, eles são monitorados em tempo de execução, resultando em sobrecarga. Então, é crucial para a análise eficiente de um sistema, que os falso-positivos sejam descobertos e eliminados.

Há também propostas que geram testes dinamicamente usando execução simbólica [14], [15]. A estratégia de tais trabalhos é usar entradas simbólicas para gerar testes para o sistema. Assim que um problema é detectado, o teste é salvo, dessa forma o desenvolvedor pode repeti-lo para corrigir o defeito encontrado. Essas propostas têm um alto custo computacional o que leva a geração dos testes ser interrompida antes que todos os casos de testes sejam gerados.

As propostas mencionadas acima realizam uma análise intra-programa, i.e., o alvo da análise é o código fonte ou binário de um único programa. Por esta razão, para analisar um sistema distribuído como um todo, usando essas abordagens é necessários analisar cada programa separadamente e os resultados globais devem ser inferidos a partir dos resultados parciais. Como não há uma visão holística do sistema, mais falso-positivos são encontrados na análise estática e mais código deve ser instrumentado na análise dinâmica. Já na execução simbólica, a falta dessa visão global faz com que nem todos os cenários de testes distribuídos sejam gerados corretamente. Defendemos que o sistema distribuído deve ser analisado como uma aplicação única para que seja possível cruzar as informações dos vários processos que constituem o sistema. Então, mais descobertas sobre a segurança de um sistema podem ser realizadas. Contudo, análise simultânea de programas não é uma tarefa trivial, pois a análise do fluxo de informação entre as partes pode ser computacionalmente ineficiente. O objetivo do nosso trabalho é fazer essa verificação cruzada de forma eficiente. Nossa proposta analisa o código de um sistema distribuído como um todo.

É importante mencionar que existem propostas para redes limitadas que analisam códigos de sistemas distribuídos baseadas em execução simbólica. Entre elas, destacamos o trabalho de Sasnauskas*et al.* [18] e *et al.* [19], Kleenet e T-Check, respectivamente. Essas ferramentas executam sistema

simbolicamente a procura de defeitos de software em geral (bugs) e permitem a exploração automática de caminhos de execução em aplicações distribuídas. Se uma asserção falha, a ferramenta registra o caso de teste para que o cenário possa ser repetido. Mas, nessas soluções o desenvolvedor deve marcar variáveis para serem simbólicas e deve escrever asserções sobre o estado do sistema. Esse passo é manual e requer conhecimento sobre a lógica da aplicação e estruturas de dados. Assim, a complexidade da solução depende das entradas simbólicas e do número de nós. Os autores de Kleenet, por exemplo, reportaram que mesmo com entradas simbólicas pequenas e poucos nós, algumas aplicações geram milhares de caminhos de execução. Nossa solução não requer que o código seja modificado, baseia-se em análise estática e, por isso, requer menos tempo de análise e menor recurso computacional.

Fluxos de Dados Contaminados. Em nosso estudo de caso detectamos fluxos de dados contaminados - caminhos entre fontes e sorvedouros que podem ser usados para atacar um sistema - em nosso caso procuramos caminhos entre entradas de usuários (*fontes*) e arranjos (*sorvedouros*). Há várias propostas na literatura que focam na detecção de fluxos de dados contaminados. Destacamos as especializadas em aplicações Web que focam na detecção de fluxos de dados contaminados para saneá-los [20] - [22]. Esses estudos são direcionados a dependências entre fontes de saídas sensíveis apenas para verificar se esses caminhos são *livres de contaminação*, ou seja, se há algum tipo de verificação ou transformação de código que previne *strings* de alcançarem sorvedouros sensíveis. Apesar desses trabalhos focarem em aplicações de rede, nenhum tratamento especial é realizado com relação a funções que fazem acesso à rede. Cada programa é analisado individualmente e qualquer informação recebida pela rede é considerada contaminada. Através das técnicas apresentadas neste artigo, é possível reduzir o número de caminhos considerados contaminados e, dessa forma, reduzir o custo da proteção de tais sistemas. Tal redução é importante para sistemas distribuídos em geral, mas é essencialmente importante para sistemas da IoT onde recursos limitados de dispositivos normalmente impossibilitam que soluções de segurança desenhadas para sistemas convencionais sejam aplicadas nesse contexto.

Grafos de Fluxo de Controle e Dependência. Várias estruturas de dados são usadas para análise de código. Entre elas, duas são especialmente importantes no contexto desse artigo: o Grafo de Controle de Fluxo (CFG) [8] e o Grafo de Dependência (DG) [9], nós discutimos sobre eles na Seção 2. Nós propomos maneiras de estender essas estruturas de dados para representar o sistema como um todo e, através de uma visão holística da aplicação, reduzir os falso-positivos comuns em análises estáticas. Como proposto neste trabalho, alguns trabalhos nas áreas de computação paralela fazem a união de CFGs [22], [23]. O objetivo dessas propostas é reduzir o número de mensagens enviadas entre processadores e então otimizar a comunicação entre eles. É possível, em alguns casos, no contexto da computação paralela, agregar múltiplos envios sem mudar a semântica de um programa para atingir uma performance melhorada. Em nosso caso, o objetivo é outro. Nós propomos um algoritmo para unir os CFGs de programas distintos com o objetivo de detectar vulnerabilidades no código. Em nosso contexto os algoritmos propostos anteriormente não se aplicam, pois precisamos saber em que fluxo está cada SEND ou RECV para determinar se ele está contaminado ou não.

VI. Conclusão

A IoT requer soluções de segurança criadas exclusivamente para seu contexto. É um fato que existem propostas de segurança para prevenir a IoT contra ataques de adversários externos. No entanto, o mesmo nem sempre ocorre para ataques realizados por adversários que conhecem o código do programa e tem acesso às entrada do sistema. As propostas de segurança de código, em particular proteções de buffer overflow, são desenhadas no contexto de redes tradicionais de computadores e não consideram as particularidades da IoT. Dessa forma não são completamente apropriadas para elas. Nesse artigo apresentamos uma solução de segurança desenhada especialmente contra ataques direcionados à exploração de vulnerabilidades de código, em especial ataques por Buffer Overflow.

Nossa proposta emprega uma abordagem pioneira para encontrar vulnerabilidades. Nessa proposta gera-se o Grafo de Fluxo de Controle e Grafo de Dependência do sistema distribuído como um todo. Assim, o analisador estático pode ter uma visão holística do sistema sob análise. Torna-se possível, então, o cruzamento de dados de programas diferentes do sistema, o que permite tornar a solução menos conservadora e mais eficiente. Os resultados indicam ser possível prover as mesmas garantias que mecanismos tradicionais de segurança, mas de maneira 83% mais eficiente em relação à quantidade de verificações de limites de arranjos a serem inseridos no código.

Referências

[1] L. Atzori, A. Iera, and G. Morabito, “The Internet of Things: A survey”, *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[2] ProofPoint, “Proofpoint uncovers internet of things (IoT) cyberattack”, http://www.proofpoint.com/about-us/press-releases/01162014.php, 2014.

[3] W. G. Halfond and A. Orso, “Amnesia: analysis and monitoring for neutralizing sql-injection attacks,” in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*.ACM, pp. 174–183, 2005.

[4] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, “TinyOS: An operating system for wireless sensor networks,” in *Ambient Intelligence*, Weber, Rabaey, and Aarts, Springer-Verlag, 2004.

[5] A. Dunkels, B. Gronvall, and T. Voigt, “Contiki - a lightweight and flexible operating system for tiny networked sensors,” in *IEEE International Conference on Local Computer Networks (LCN’04)*, 2004, pp. 455–462.

[6] B. Chess and J. West, *Secure Programming with Static Analysis*, Addison-Wesley Professional, 2007.

[7] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “Addresssanitizer: a fast address sanity checker,” in *USENIX*, 2012, pp. 28–28.

[8] F. E. Allen, “Control flow analysis,” *SIGPLAN Not.*, vol. 5, pp. 1–19, 1970.

[9] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe, “The program dependence web: a representation supporting control-data and demand-driven interpretation of imperative languages,” in *PLDI*, ACM, 1990, pp. 257–271.

[10] C. Lattner and V. S. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *CGO*, 2004, pp. 75–88.

[11] M. Bishop, S. Engle, D. Howard, and S. Whalen, “A taxonomy of buffer overflow characteristics,” *Dependable and Secure Computing, IEEE Transactions on*, vol. 9, no. 3, pp. 305–317, 2012.

[12] W. Dietz, P. Li, J. Regehr, and V. Adve, “Understanding integer overflow in c/c++,” , 2012, pp. 760–770.

[13] R. E. Rodrigues, V. H. S. Campos, and F. M. Q. Pereira, “A fast and low overhead technique to secure programs against integer overflows,” in *CGO ACM*, 2013.
[14] T. Wang, T. Wei, Z. Lin, and W. Zou, “Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution,” in *Proc. of Network and Distributed System Security Symposium (NDSS)*.1em plus 0.5em minus 0.4emCiteseer, 2009.
[15] D. Molnar, X. C. Li, and D. A. Wagner, “Dynamic test generation to find integer bugs in x86 binary linux programs,” in *Proceedings of the 18th conference on USENIX security symposium*.1em plus 0.5em minus 0.4emUSENIX Association, 2009, pp. 67–82.
[16] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, “Securing web application code by static analysis and runtime protection,” in *Proceedings of the 13th international conference on World Wide Web*.1em plus 0.5em minus 0.4emACM, 2004, pp. 40–52.
[17] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, “Saner: Composing static and dynamic analysis to validate sanitization in web applications,” in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*.1em plus 0.5em minus 0.4emIEEE, 2008, pp. 387–401.
[18] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, “Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment,” in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, ACM, 2010, pp. 186–196.
[19] P. Li and J. Regehr, “T-check: bug finding for sensor networks,” in *9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, ACM, 2010, pp. 174–185.
[20] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, “MODIST: Transparent model checking of unmodified distributed systems,” in *USENIX Symposium on Networked Systems Design and Implementation NSDI’09*, 2009, pp. 213–228.
[21] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, “Prospex: Protocol specification extraction,” in *IEEE Symposium on Security and Privacy*, 2009, pp. 110–125.
[22] G. Bronevetsky, “Communication-sensitive static dataflow for parallel message passing applications,” in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2009, pp. 1–12.
[23] B. Di Martino, A. Mazzeo, N. Mazzocca, and U. Villano, “Parallel program analysis and restructuring by detection of point-to-point interaction patterns and their transformation into collective communication constructs,” *Science of Computer Programming*, vol. 40, no. 2, pp. 235–263, 2001.
[24] SIoT Repository. https://code.google.com/p/ecosoc/wiki/SIoT.
[25] Teixeira, F. A., Pereira, F., Vieira, G., Marcondes, P., Wong, H. C., Nogueira, J. M. S., & Oliveira, L. B. SIoT – Defendendo a Internet das Coisas contra Exploits. *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC*), 2014.
[26] Karlof, C. and Wagner, D. (2003). Secure routing in wireless sensor networks: Attacks and countermeasures. In *1st IEEE Int’l Workshop on Sensor Network Protocols and Applications*.
[27] Wangham, M. S., Domenech, M. C., and de Mello, E. R. (2013). Infraestrutura de autenticação e de autorização para internet das coisas. *In Minicursos, volume 1 of 13th Brazilian Symposium on Information and Computer System Security (SBSeg’13)*. SBC.
[28] Casado, L., & Tsigas, P. (2009). “Contikisec: A secure network layer for wireless sensor networks under the contiki operating system.” *In Identity and Privacy in the Internet Age (pp. 133-147).* Springer Berlin Heidelberg.

Leonardo B. Oliveira é Professor Adjunto do Departamento de Ciência da Computação da Universidade Federal de Minas Gerais (UFMG). Ele é bacharel (UFMG), mestre (UFMG) e doutor (Unicamp) em Ciência da Computação. Trabalhou junto ao Information Security Group da Universidade de Londres. É membro da Comissão Especial de Segurança da Informação e Sistemas Computacionais da Sociedade Brasileira de Computação (SBC). Há vários anos Leonardo participa do comitê de programa do SBSeg e foi um dos organizadores do simpósio na edição de 2009. Leonardo já foi agraciado com os prêmios CTDSeg e CTD da SBC (ambos na categoria doutorado), bem como os prêmios IEEE Young Professional Award e Microsoft PhD Fellowship Award. Leonardo possui dezenas de trabalhos científicos publicados na área de criptografia aplicada e segurança da informação.

José Marcos Silva Nogueira é professor Titular no Departamento de Ciência da Computação da UFMG. É engenheiro eletricista pela Universidade Federal de Minas Gerais - UFMG (1975), mestre em Ciência da Computação pela UFMG (1979) e doutor em Engenharia Elétrica pela Universidade Estadual de Campinas - Unicamp (1985). Realizou pós-doutorado na University of British Columbia - UBC, Canadá (1988-89) e passou ano sabático na Université Pierre et Marie Curie (Univ. Paris 6), França (2004-05). Foi chefe do DCC/UFMG, coordenador do Programa de Pós-Graduação e do curso de bacharelado em Ciência da Computação da UFMG. Suas áreas de interesse são redes de computadores, gerenciamento de redes, redes de sensores sem fio, computação móvel e redes tolerantes a interrupções. Tem experiência em projetos de desenvolvimento de software, particularmente de gerenciamento e supervisão de redes. Orientou em torno de 40 alunos de mestrado e oito alunos de doutorado

Fernando M. Q. Pereira recebeu o título de doutor em Ciência da Computação pela Universidade da Califórnia, Los Angeles, em 2008. Atualmente ele é professor adjunto no Departamento de Ciência da Computação (DCC) da Universidade Federal de Minas Gerais (UFMG), onde ensina linguagens de programação e análise estática de programas. Fernando tem pesquisado maneiras de construir compiladores que gerem código mais seguro. Desse esforço resultaram diversas melhorias em compiladores de grande popularidade. Um exemplo é a análise que previne a injeção de código SQL usada por PHC, o compilador de PHP.

Hao Chi Wong é pesquisadora de segurança da Intel, com trabalho focado na segurança de produtos. Tem trabalhos e publicações em diversas áreas de segurança, incluindo: protocolos de segurança, segurança em redes sem fio e redes de sensores, segurança usável, métodos formais aplicados à segurança, e segurança de hardware. Foi pesquisadora do Xerox PARC, e professora visitante no Departamento de Ciência da Computação (DCC) da Universidade Federal de Minas Gerais (UFMG). Doutora pela Carnegie Mellon University, mestre pelo Departamento de Informática (DI) da Pontifícia Universidade Católica (PUC) do Rio de Janeiro, e bacharel pelo DCC/UFMG.

Fernando Augusto Teixeira é professor da Universidade Federal de São João del-Rei (UFSJ) e doutorando em Ciência da Computação pela Universidade Federal de Minas Gerais (UFMG), onde obteve o título de mestre em Ciência da Computação em 2005 e graduou-se em 2000. Em 2009 obteve o título de MBA em Gestão Estratégica da Tecnologia da Informação pela Fundação Getúlio Vargas (FGV). Atuou por 10 anos em Fábrica de Software, principalmente, em projetos voltados para o setor de telecomunicações. Tem interesse por sistemas distribuídos, redes sem fio, compiladores e segurança.

Gustavo Vieira Machado é estudante de graduação em Engenharia de Controle e Automação pela Universidade Federal de Minas Gerais (UFMG) e aluno de Iniciação Científica do Wireless, Innovative, Security, and Embedded Systems & Models, Algorithms and Protocols – WISEMAP. Suas principais áreas de interesse são segurança, sistemas embarcados e sistemas distribuídos.

Pablo Marcondes Fonseca é estudante de graduação de Ciência da Computação (2011-presente) e pesquisador da Universidade Federal de Minas Gerais (UFMG). Pablo é membro do projeto Intel Energy-Efficient Instrumentation to Secure Systems-on-a-Chip Devices (eCoSoC). Suas principais áreas de interesse são Arquitetura de Software, Segurança, sistemas embarcados e sistemas distribuídos.