Siot – Defendendo a Internet das Coisas contra Exploits

Fernando A. Teixeira[1], **Fernando Pereira**[1], **Gustavo Vieira**[1], **Pablo Marcondes**[1], **Hao Chi Wong**[2], **José Marcos S. Nogueira**[1], **Leonardo B. Oliveira**[1]

[1]Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG, Brazil

{fateixeira,fernando,gvieira,pablom,jmarcos,leob}@dcc.ufmg.br

Intel Corporation – Santa Clara, CA

hao-chi.wong@intel.com

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, SIoT, makes use of a pioneering solution to pinpoint vulnerabilities: we cross-check data from communicating nodes. SIoT provides the same guarantees as traditional security mechanisms, but it is about 83% more efficient, according to the experiments that this article describes.*

Resumo. *A Internet das Coisas (IoT – Internet of Things) requer soluções de segurança feitas sob medida. Embora haja propostas que tratem ataques externos de forma satisfatória, o mesmo não ocorre para ataques internos. Sobre esses últimos, as propostas existentes nem sempre consideram as características de IoT e, portanto, não lhes são adequadas. O objetivo deste trabalho é conceber uma solução para proteger sistemas de IoT contra ataques internos. Nossa solução, SIoT, emprega uma abordagem pioneira na busca de vulnerabilidades: o cruzamento de dados de diferentes nós da rede. SIoT provê as mesmas garantias que mecanismos de segurança tradicionais, porém é cerca de 83% mais eficiente, de acordo com os experimentos que este artigo descreve.*

1. Introdução

A Internet das Coisas (IoT – *Internet of Things*) é uma infraestrutura de rede dinâmica e global com capacidades de autoconfiguração, baseada em protocolos de comunicação padronizados e interoperáveis, onde "coisas" físicas e virtuais têm identidades, atributos físicos e personalidades virtuais. Usam interfaces inteligentes bem como são naturalmente integradas à Internet [Atzori et al. 2010]. Na IoT, as *"coisas"* ou objetos devem se tornar participantes ativos em processos de negócio, informacionais e sociais, onde serão capazes de interagir e comunicar entre elas mesmas, trocar informações coletadas do ambiente, reagindo autonomamente aos eventos do mundo físico real, bem como influenciar esse contexto sem intervenção direta do ser humano.

As interfaces na forma de serviço facilitam as interações entre essas *"coisas inteligentes"* via Internet e permitem a consulta de informações e troca de estados associados às "coisas". Mas questões como segurança e privacidade devem ser consideradas. É nesse contexto que se insere o presente trabalho: a comunicação necessária entre as "coisas" ou objetos, efetivamente realizadas através das interfaces de rede, abre possibilidades de ataques externos ou internos de segurança. Os atacantes buscam então explorar vulnerabilidades para poderem influenciar o comportamento ou obter o controle do sistema.

A Internet das Coisas (IoT – Internet of Things) requer soluções de segurança feitas sob medida [Wangham et al. 2013]. Isso porque, ao contrário de um computador tradicional, os elementos de IoT usualmente: (i) são dotados de menor capacidade de processamento, memória e energia [Atzori et al. 2010]; (ii) realizam tarefas de forma colaborativa [Atzori et al. 2010]; e (iii) executam sistemas eminentemente desenvolvidos usando a linguagem C ou linguagens nela baseadas (TinyOS [Levis et al. 2004], ContikiOS [Dunkels et al. 2004] e Linux embarcado, por exemplo). Entretanto, as propostas de segurança existentes nem sempre levam em conta tais características e, portanto, nem sempre são adequadas para a IoT.

Em relação a ataques disparados por adversários externos (*outsiders*), por exemplo, já existem propostas que atendem a IoT satisfatoriamente ([Oliveira et al. 2006, Simplício et al. 2010], por exemplo). Algumas delas são capazes de garantir propriedades como sigilo e autenticação de mensagens a um custo aceitável mesmo para a IoT [Karlof and Wagner 2003]. Assim, ataques de espionagem, personificação (*spoofing*) e retransmissão (*replay*) podem ser eficientemente evitados [Karlof and Wagner 2003].

No entanto, para algumas questões de segurança ainda não há propostas personalizadas para IoT. Um exemplo são alguns ataques disparados por adversários internos (*insiders*), ou seja, por aqueles cujas credenciais de acesso são válidas. As mensagens trocadas por esses adversários são legítimas na medida em que elas são autenticadas com sucesso pelos seus destinatários. Isso, contudo, não as torna menos maliciosas. Essas mensagens objetivam, por exemplo, explorar vulnerabilidades de sistemas para então obter o seu controle; ou seja, elas são, no fundo, *exploits* [Chess and West 2007]. Para essa classe de ataques, ainda não há soluções plenamente adequadas para o contexto de IoT.

Objetivo. O objetivo deste trabalho é o estudo da questão de segurança de IoT e proposição de soluções que envolvem algoritmos, técnicas e mecanismos para proteção dos módulos de IoT contra ações de exploração de vulnerabilidades de código (*exploits*). Mais precisamente, ela objetiva conceber uma solução de segurança de código para IoT capaz de defender seus sistemas contra ataques internos à rede, em especial contra Buffer Overflows. Nossa proposta, *SIoT* (*Secure IoT*), emprega uma abordagem pioneira na busca de vulnerabilidades: o cruzamento de dados de diferentes módulos de um mesmo sistema distribuído[1]. Esse cruzamento de dados resulta em mais informação e, portanto, 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.

Contribuição. As principais contribuições deste trabalho são as seguintes. **(i)** Indicação das características de IoT que mais influenciam no projeto de soluções de

[1] Aqui, módulos são as partes do sistema distribuído que são executadas por diferentes nós da rede.

segurança; **(i)** Projeto de uma solução personalizada para IoT, que utiliza uma técnica inédita para apontar vulnerabilidades. **(iii)** Concepção de um algoritmo capaz de unir, eficientemente, grafos de diferentes módulos de um sistema. **(iv)** Desenvolvimento de uma ferramenta disponível *online*[2] que implementa nossa solução. **(v)** Avaliação da solução utilizando o *ContikiOS* [Dunkels et al. 2004], um sistema operacional para a IoT.

Acerca deste último ponto, em especial, utilizamos a SIoT para proteger as aplicações `udp-ipv6` e `webser/wget` do ContikiOS contra ataques de *Buffer Overflow*. Os resultados indicam que para um mesmo nível de segurança, a SIoT se mostra pelo menos 83% mais eficiente que por propostas hoje amplamente utilizadas.

Organização do texto. A Seção 2 discorre sobre as características de IoT que mais impactam a segurança. O modelo de ataque e os conceitos de segurança de código utilizados no artigo são apresentado na Seção 3. A Seção 4 descreve nossa solução. A avaliação da solução via um estudo de caso e resultados é apresentado na Seção 5. Os trabalhos relacionados são discutidos na Seção 6 e as conclusões na Seção 7.

2. IoT: Características & Segurança

A IoT se difere das redes de computadores tradicionais em inúmeros aspectos [Atzori et al. 2010]. Por exemplo, sua escala é maior, sendo composta por centenas ou milhares de nós. Os avanços na miniaturização permitem que as coisas menores tenham cada vez mais a capacidade de interagir e se conectar [Wangham et al. 2013].

Contudo, são as características que impactam a segurança que mais nos importam (Tabela 1). Neste contexto, é na escassez de recursos que reside a diferença chave entre IoT e redes tradicionais [Wangham et al. 2013]. Idealmente, os nós da IoT são compostos por Sistemas Embarcados (SEs) de baixo custo monetário, a fim de reduzir o valor final de produtos. Dessa forma, SEs possuem poucos recursos computacionais (processamento, memória, energia, etc.) e são incapazes de executar sistemas "pesados". Para mitigar essa questão, sistemas para IoT são frequentemente desenvolvidos em linguagens de programação eficientes, tal como C.

O problema é que sistemas desenvolvidos em C são inerentemente mais inseguros [Chess and West 2007]. A linguagem C torna os sistemas mais "leves" e, por isso, mais apropriados a SEs. No entanto, tal eficiência tem preço: sistemas desenvolvidos em C são comumente mais vulneráveis. Uma das razões da eficiência de C está no fato de que a linguagem não realiza certas verificações que Java, por exemplo, realiza[3]. Uma dessas verificações é a de limites de arranjos; e é justamente aí que surge o perigo do ataque de *Buffer Overflow – BOF* [Chess and West 2007], um dos ataques mais comuns a sistemas computacionais. E mecanismos de segurança para mitigar ataques de BOF presentes em PCs[4], muitas vezes não estão presentes em SEs.

Outro fator que impacta a segurança de forma preponderante é que tarefas na IoT são usualmente realizadas de forma colaborativa [Atzori et al. 2010]. Em virtude disso, a natureza dos sistemas executados é alterada, isto é, eles passam a ser eminentemente distribuídos. Isso, naturalmente, incrementa sua complexidade o que, por sua vez, torna a

[2]https://bitbucket.org/ecosoc/siot

[3]Formalmente, a ausência dessas verificações é porque C é fracamente tipada (*weakly typed*).

[4]Por exemplo o NOEXEC `http://pax.grsecurity.net/docs/noexec.txt`

Características	Redes Tradicionais	IoT
Escala	dezenas ou centenas de nós	centenas ou milhares de nós
Linguagem de Programação	Python, Java, C, C++ e C#	Eminentemente C e C++
Realização Tarefas	independentemente	colaborativamente
Recursos Computacionais	"rico"	"pobre"
Recurso Crítico	tempo ou memória	energia
Arquitetura dos Nós	64,32 *bits*	8, 16, ou 32 *bits*

Tabela 1. Redes Tradicionais versus IoT: aspectos que impactam a segurança.

busca por vulnerabilidades mais desafiadora. Na medida em que os nós colaboram, eles também trocam mais mensagens e, portanto, aumenta-se a janela para potenciais ataques.

Por fim, a arquitetura dos nós de IoT também influência a segurança. Diferentemente de PCs, em que hoje a arquitetura padrão é de 64 *bits*, SEs são geralmente baseadas em arquiteturas de de 8, 16, ou 32 *bits*. Essa diferença altera a forma com a qual o *hardware* executa um sistema e, portanto, impacta a segurança [Chess and West 2007].

3. Conceitos Fundamentais

Nessa seção, descrevemos nossas premissas e modelo de ataque (Seção 3.1) e apresentamos os conceitos de segurança de sistemas (Seção 3.2) utilizados no artigo.

3.1. Premissas & Modelo de Ataque

Grande parte de ataques externos podem ser evitados com o uso de cifras (*ciphers*) e códigos de autenticação de mensagens [Karlof and Wagner 2003]. Assim, já há na literatura diversas propostas capazes de proteger uma rede contra essa classe de ataques – por exemplo, SSL/TLS[5] para comunicação fim-a-fim em redes tradicionais; e, por exemplo, o NEKAP [Oliveira et al. 2006], que fornece segurança na camada de enlace para redes *ad hoc* com escassez de recursos como a IoT. Neste modelo, quando uma mensagem é recebida, verifica-se sua validade e ela só é tratada pelo sistema caso a verificação seja bem sucedida. Tais propostas dificultam a ação de adversários que busquem forjar as mensagens do sistema. Nosso trabalho pressupõe que esses mecanismos tiveram sucesso e se concentra nos ataques realizados por adversários que conseguiram acesso ao sistema.

Consideramos que o adversário possui acesso às entradas dos dispositivos de rede e, portanto, logra com que outros membros da rede validem suas mensagens [Karlof and Wagner 2003]. Como o adversário possui os mesmos privilégios de um usuário comum ele pode realizar ataques internos e, por isso, a efetividade das propostas citadas no parágrafo anterior é reduzida. É necessário, portanto, que mecanismos de segurança protejam o sistema contra ataques que exploram falhas no código através da manipulação das entradas dos sistema.

Em particular, nossa solução pressupõe que: (i) as mensagens trocadas na rede são autenticadas; e (ii) sistemas executados pelos nós são conhecidos e estão íntegros. Assim, nosso adversário tem acesso às "coisas" como um usuário comum (*ordinary user*), mas não é capaz de trocar seu código ou injetar mensagens falsas diretamente no canal de rede.

[5] `http://tools.ietf.org/html/rfc5246`

Entretanto, o adversário é capaz de entrar com dados que levem o programa a um estado inconsistente e, dessa maneira, até mesmo fazer com as aplicações enviem mensagens espúrias para outros nós da rede. Esse cenário é bastante comum: em geral, os adversários primeiro obtêm privilégios menores para então obterem maiores privilégios. E como as "coisas inteligentes" possuem outros canais de interação com o mundo, como sensores ou controles remotos, é possível que o adversário manipule essas entradas de forma a induzir os 'coisas" a erros. Por exemplo, uma sequência de teclas em um controle remoto de uma TV inteligente pode ser usada para explorar uma falha de código e fazer com que a TV envie mensagens não desejadas na rede ou um isqueiro aceso perto de um sensor de temperatura pode ser usado para explorar uma falha de *integer overflow* no código.

3.2. Segurança de Sistemas

A maioria das propostas de defesa para sistemas computacionais são baseadas na Análise Estática [Chess and West 2007], na Análise Dinâmica [Serebryany et al. 2012], ou na combinação de ambas. A concepção dessas propostas, no entanto, é uma tarefa desafiadora já que qualquer propriedade não trivial de linguagens recursivamente enumeráveis é um problema indecidível. Ou, em outras palavras, não existe programa genérico capaz de decidir se um outro programa qualquer é ou não vulnerável.

Na Análise Estática [Chess and West 2007] o sistema é inspecionado antes do programa ser implantado (*deployed*). Por essa razão, é também conhecida como *análise de código*. A vantagem do método é que não há sobrecarga (*overhead*) durante a execução do sistema. Seu aspecto negativo é que a análise não possui informações que só estarão disponíveis em tempo de execução. Não raro tal "desinformação" impossibilita afirmar se há ou não uma vulnerabilidade em certos trechos do sistema. Na dúvida, a análise é conservadora e, assim, presume-se que a vulnerabilidade existe. Esse conservadorismo, por sua vez, traduz-se em falso-positivos[6].

Na Análise Dinâmica [Serebryany et al. 2012], por outro lado, o sistema é, sim, executado. Agora, a técnica pode tirar proveito das informações só disponíveis em tempo de execução. Isso, por um lado, mitiga a questão dos falso-positivos, comuns à Análise Estática. No entanto, seus resultados são pertinentes apenas às entradas testadas e, assim, não se pode tirar conclusões acerca do comportamento geral do programa.

Em razão das naturezas complementares das análises Estática e Dinâmica, é comum o emprego da Análise Híbrida, ou seja, a combinação de ambas. Usualmente, a Análise Estática é primeiro empregada para apontar-se as vulnerabilidades e, posteriormente, a Análise Dinâmica instrumenta o sistema para monitorá-lo em tempo de execução, em seus trechos supostamente vulneráveis. E eis o motivo pelo qual falso-positivos acarretam sobrecarga. Embora eles não apresentem perigo, eles são monitorados em tempo de execução, o que resulta em sobrecarga. Portanto, é fundamental para a eficiência de um sistema que os falso-positivos sejam descobertos e eliminados.

Diversos ferramentais são utilizados durante a análise de sistemas. Dentre elas, duas são especialmente importantes no contexto deste trabalho: os Grafos de Fluxo de Controle (CFG) e os Grafos de Dependências [Chess and West 2007]. O CFG é um grafo dirigido que representa as possíveis sequências de instruções que são processadas durante

[6]Posteriormente, mostraremos porque falso-positivos são particularmente indesejáveis na IoT

a execução de um programa. Um vértice do CFG é chamado um bloco básico. Um bloco básico é um conjunto maximal de instruções que sempre são executadas em sequência. Assim, um bloco básico possui um desvio, seja ele condicional ou não, somente no final da lista de suas instruções constituintes. Existe uma aresta entre um bloco básico b_1 e um bloco básico b_2 se, e somente se, o programa pode fluir de b_1 para b_2. O grafo de dependência, por sua vez, possui um nó para cada variável e cada operação do programa. O grafo de dependências também é direcionado. Nele, há uma aresta entre cada variável u e uma instrução i se i representa uma instrução que usa u.

4. Siot

A fim de prover ao analisador estático uma visão holística do programa sob análise, este artigo introduz o conceito de 'Grafo de Controle de Fluxo de Sistemas Distribuídos', descrito na Seção 4.1. Em seguida, mostramos a arquitetura da Siot na Seção 4.2.

4.1. Grafo de Controle de Fluxo de Sistemas Distribuídos

Grafos de fluxo de controle têm, por quase cinco décadas, desfrutado de um papel central na análise de programas. Por outro lado, neste artigo clamamos que essa representação não é suficientemente expressiva para analisar programas distribuídos. A fim de contornar essa limitação, propomos uma forma de criar um Grafo de Fluxo de Controle Distribuído (*Distributed Control Flow Graph* – DCFG). Esse grafo conecta, em uma única representação, os CFGs dos vários programas que constituem uma aplicação distribuída.

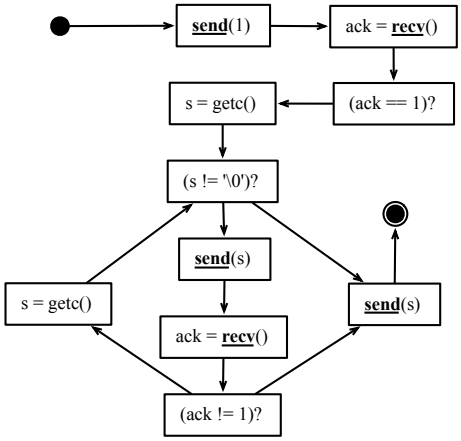
Cada módulo de um sistema distribuído possui seu próprio CFG. A Figura 1 mostra um sistema *echo*, com módulos cliente e servidor e seus respectivos CFGs. Na linha 5 do módulo servidor (Figura 1b) temos um exemplo de uma vulnerabilidade. Nessa linha, o arranjo `msg` recebe dados da rede via um `recv`. Esse `recv` recebe dados do segundo `send` do módulo cliente (linha 6 da Figura 1a). E esse `send`, por sua vez, depende do `getc` (linha 4) desse mesmo módulo. Logo há um caminho entre o usuário e o arranjo `msg`, o que o torna vulnerável.

Mas também temos um exemplo de um falso-positivo nesse sistema. A primeira vista, se analisarmos apenas o módulo servidor (Figura 1b) concluímos que o arranjo `msg` é vulnerável por depender do primeiro `recv` (linha 1 da Figura 1b). Mas ao analisar o módulo cliente (Figura 1a) verifica-se que esse `recv` só recebe informações do primeiro `send` (linha 1 da Figura 1a). Entretanto, esse `send` envia apenas uma constante (`send(1)`). Portanto, não há dependências com o usuário. Deste exemplo, concluímos que a análise de um sistema distribuído pode beneficiar-se de um DCFG.

Interessa-nos, portanto, uma maneira de unir CFGs de módulos individuais, de maneira tal que as análises já descritas na literatura possam ser aplicadas sobre o DCFG resultante sem modificações. A solução trivial desse problema é unir todos os `sends` com todos os `recvs` de um sistema. Entretanto, se assim o procedermos várias arestas desnecessárias serão incluídas. São desnecessárias as arestas que ligam um send s_A de um módulo A a um receive r_B de um módulo B sem no entanto haver um cenário que ao mesmo tempo leve o módulo A a s_A e o módulo B a s_B. Vejamos novamente a Figura 1. Como exemplo, podemos citar que é desnecessário adicionar uma aresta entre o `A:send` e o `H:recv` porque a mensagem enviada por A será recebida por `F:recv` e não por `H:recv`. Como não há um caminho que permita que a mensagem enviada pelo

(a)

```
1  send(1);
2  ack = recv()
3  if (ack == 1) {
4      s = getc();
5      while (s != '\0') {
6          send(s)
7          ack = recv();
8          if (ack != 1) {
9              break;
10         } else {
11             s = getc();
12         }
13     }
14     send(s);
15 }
```



(b)

```
1  msg = recv();
2  if (msg == 1) {
3      send(1);
4      do {
5          msg = recv();
6          putc(msg);
7          if (msg != '\0')
8              send(1);
9          else
10             break;
11     } while (1);
12 } else {
13     send(0);
14 }
```

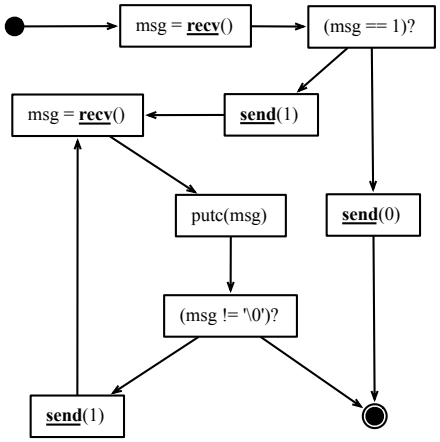


Figura 1. Grafo de fluxo de controle de dois módulos do sistema *Echo*. (a) Módulo cliente. (b) Módulo servidor *echo*.

`A:send` chegue a `H:recv`, uma aresta inserida entre esses nodos seria desnecessária. Para resolver esse problema, propomos um algoritmo que leva em consideração o CFG de cada módulo e como esses módulos interagem via rede de forma a reduzir a quantidade de arestas desnecessárias.

Com o objetivo de encontrar um DCFG melhor que a solução trivial, introduzimos a noção de nível. A Equação 1 define os níveis de um `send` ou `recv` no grafo como segue:

$$\begin{array}{rcl} \mathit{nível}(cfg, 0) & = & \{\mathit{root}\} \\ \mathit{nível}(cfg, n) & = & \{v \mid \overrightarrow{uv} \in cfg \wedge u \in \mathit{level}(cfg, n-1)\} \end{array} \quad (1)$$

Em outras palavras, consideremos inicialmente a definição de níveis dos `sends`. O nível zero (L_0) possui a raiz do grafo. O nível um (L_1) contém os `sends` sucessores diretos da raiz. O nível dois (L_2) contém os `sends` sucessores diretos de algum `send` do nível L_1. E assim recursivamente. O algoritmo termina quando alcança o fim do programa ou quando um nível possui o mesmo conjunto de `sends` de um nível anterior. O mesmo raciocínio se aplica aos `recvs`. A Figura 2 mostra os conjuntos de níveis dos `sends` do cliente *echo* (esquerda) e `recvs` do servidor (direita).

A partir da definição dos níveis, concluímos que devemos incluir no grafo uma

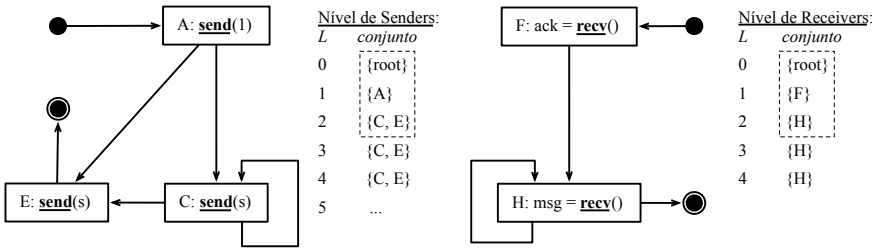


Figura 2. Conjuntos de níveis dos sends do cliente *echo* (esquerda) e recvs do servidor (direita). A caixa tracejada delimita os níveis distintos.

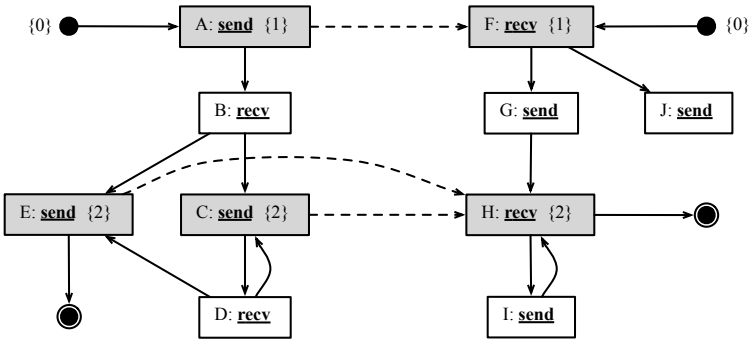


Figura 3. Ligações entre sends do cliente *echo* e recvs do servidor *echo* do nosso exemplo.

aresta entre "*A:send(1)*" (s_A) e "*F:recv(1)*" (r_F) pois esses nós pertencem ao nível L_0. Ou seja, teremos uma aresta $\overrightarrow{s_A r_F}$. Também devemos incluir as arestas $\overrightarrow{s_C r_H}$ e $\overrightarrow{s_E r_H}$ pois esses nós pertencem ao nível L_1. Não é necessário criar as arestas $\overrightarrow{s_A r_H}$, $\overrightarrow{s_C r_F}$ e $\overrightarrow{s_E r_F}$ porque esses nós pertencem a níveis diferentes. As ligações entre `sends` do cliente e `recvs` do servidor *echo* do nosso exemplo podem ser vistas na Figura 3. Os números próximos aos vértices representam o nível ao qual o nó pertence.

Grafo de Dependência de Sistemas Distribuídos. A partir do DCFG, podemos criar um grafo de dependências distribuído. Tal grafo pode ser construído a partir da união dos grafos de dependências individuais de cada módulo, via uma estratégia semelhante àquela que usamos para unir CFGs. Cada programa do sistema possui seu próprio grafo de dependência. Para cada instrução de acesso a rede, é criado um vértice no grafo para representar essa operação. O vértice é marcado como um nó de rede de leitura ou de escrita e inserido em uma lista correspondente. Insere-se, então, uma aresta de dependência entre o vértice de rede encontrado e os vértices correspondentes no outro programa de acordo com o DCFG do sistema construído anteriormente. Dessa forma, o grafo de dependências final é a união dos grafos de cada programa. Essa estrutura representa o grafo de dependências do sistema distribuído como se tal sistema fosse um único programa.

Complexidade. O algoritmo de conexão de nodos possui um pior caso de $O(2^N)$, sendo N o número de operações de envio ou de recebimento de dados do grafo de protocolo. Para fundamentar esse limite, notamos que o algoritmo termina assim que um nível repetido é encontrado. Entretanto, a quantidade de níveis é finita, e limitada pelo número

possível de subconjuntos de vértices do grafo, que é 2^N. Vale ressaltar, contudo, que não encontramos um exemplo que validasse esse pior caso. Em todos os nossos exemplos práticos, o algoritmo gerou uma quantidade de níveis inferior ao número de vértices do grafo de protocolo. Por fim, é importante observar que a complexidade da proposta não depende do número de nós da rede. Mas sim do número de programas diferentes que executam nos nós, que é pequena, pois vários executam o mesmo código.

4.2. Arquitetura e Implementação

A Siot foi implementada como uma extensão do compilador LLVM[7] [Lattner and Adve 2004] através de um conjunto de passos de otimização. A arquitetura da Siot é apresentada na Figura 4. O processo de análise é constituído por três etapas: (i) Compilação e preparação dos *bytecodes* (*.bc*) dos módulos do sistema – *Merge*; (ii) Análise das instruções do *bytecode* do sistema distribuído – *NetInputDep*, *NetLevel* e *NetDepGraph*; (iii) Análise de arranjos vulneráveis, instrumentação dos módulos e geração de estatísticas/grafo de arranjos vulneráveis – *NetVulArrays*.

Inicialmente, os módulos do sistema são compilados via *clang* (compilador do LLVM) e são gerados *bytecodes* de cada módulo. O passo *Merge* recebe os diversos módulos do sistema distribuído a ser analisado e gera como saída um *bytecode* único. O Merge analisa quais são as possíveis funções de acesso a rede e exibe para o usuário. O usuário determina quais realmente desempenham o papel de envio e recebimento de dados via rede e o Merge atualiza o arquivo de configuração – config.txt – que será utilizado nas próximas execuções. Uma vez definidas as funções de acesso à rede, são inseridas *tags* no *bytecode* para identificá-las. Também são inseridas *tags* para diferenciar cada função e variável global de cada módulo do sistema. Feito isso, é gerado um *bytecode* unificado contendo os módulos do sistema.

Em seguida, o passo *NetInputDep* avalia quais são as entradas do sistema, ou seja, quais instruções interagem com o usuário, com a rede ou com o sistema de arquivos. Nesse passo são geradas listas com todas as entradas do sistema classificadas como dependentes de rede ou não. Em paralelo, o passo *NetLevel* implementa o algoritmo de definição de níveis descrito na seção anterior (4.1). Uma vez definidos os níveis de cada interação com a rede e com auxílio das *tags* inseridas pelo Merge, o passo NetDepGraph constrói o Grafo de Dependências Distribuído.

De posse do Grafo de Dependências Distribuído, o passo *NetVulArrays* procura por caminhos entre entradas de um módulo e seus acessos à memória. O NetVulArrays verifica, também, quais são dependentes de rede e, desses, quais dependem de entradas do módulo que envia as informações via rede. Ao final, o NetVulArrays gera: (i) estatísticas de verdadeiros-positivos e falsos-positivos; (ii) grafo dos caminhos considerados vulneráveis; (iii) *bytecode* dos módulos instrumentados.

5. Estudo de Caso

BOFs (*Buffer Overflows*) são um dos ataques mais utilizados para tomada de controle de sistemas [Chess and West 2007]. Eles são evitados através de Verificações de Limites de Arranjo (*Array Bounds-Checks* – ABCs) [Chess and West 2007]. Ou seja,

[7]LLVM não é uma sigla, mas sim o nome de uma coleção de tecnologias e ferramentas de compilação modulares e reutilizáveis.

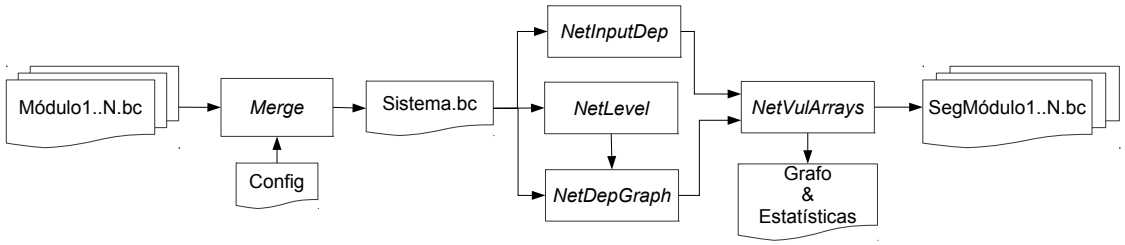


Figura 4. Arquitetura da Siot.

UDP Server

```
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 5. Trecho do Servidor udp-ipv6 do ContikiOS.

UDP Client

```
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 6. Trecho do Cliente udp-ipv6 do ContikiOS.

instrumentações no código que checam antes de acessos a memória se um índice está dentro dos limites do arranjo. Até então, contudo, a utilização de ABCs no contexto de IoT era ineficiente. Isso porque as propostas existente são conservadoras e instrumentam um número considerável de falso-positivos. Nesta seção, utilizamos a Siot para proteger sistemas de IoT contra BOFs. Em particular, avaliamos a Siot sobre as seguintes aplicações do ContikiOS: (i) `udp-ipv6` cliente e servidor e (ii) `webserver` e `wget`.

A aplicação udp-ipv6[8] demonstra como construir um cliente e um servidor UDP usando a pilha de protocolos 6LoWPAN[9]. No servidor, (Figura 5), as mensagens são recebidas via rede através da função `uip_newdata()` (linha 4) e são acessadas através do arranjo `uip_appdata`. As ferramentas tradicionais consideram esse arranjo vulnerável e, para protegê-lo, utilizam um ABC que seria acionado sempre que uma mensagem for recebida. No módulo cliente (Figura 6), da mesma forma, um outro arranjo seria protegido por ferramentas tradicionais. Isso porque a função `uip_newdata()` (linha 3) é chamada para verificar se há mensagens recebidas e, em seguida, os dados são transferidos para o arranjo `str` (linha 4).

No entanto, ao se analisar o sistema como um todo – isto é, ambos os módulos –, nota-se que os arranjos supracitados não são de fato vulneráveis. Contudo, as ferramentas tendem a reportar tal situação como uma vulnerabilidade. Esse aviso é um falso positivo.

[8]https://github.com/contiki-os/contiki/tree/master/examples/udp-ipv6

[9]IPv6 over Low power Wireless Personal Area Networks – http://tools.ietf.org/html/rfc6282

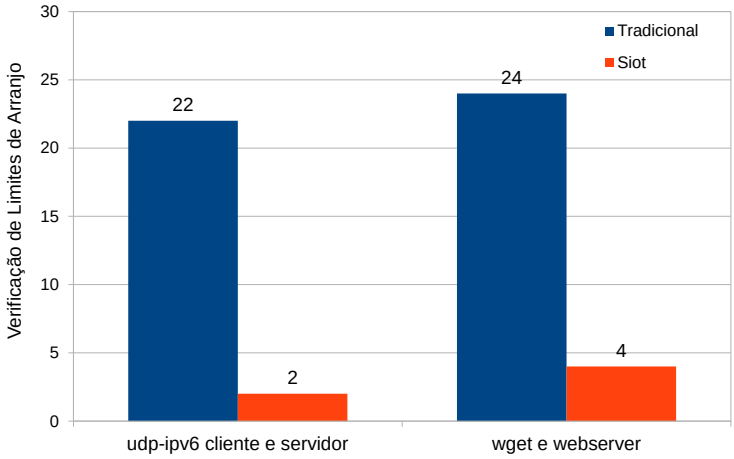


Figura 7. Verificações de Limites de Arranjos inseridos em cada aplicação.

Veja, a mensagem que o cliente prepara não possui qualquer dependência com fontes externas de dados (Figura 6, linhas 10 a 14). Ora, se tal mensagem não é vulnerável em sua origem, seguindo nosso modelo de ataque (Seção 3.1), ela também não é vulnerável no destino. No servidor, de forma análoga, a mensagem enviada para o cliente não possui dependência com entradas de dados (Figura 5, linhas de 8 a 11). Em suma, os ABCs que seriam inseridos eram, então, desnecessários.

Ao protegermos as aplicações com a Siot, reduzimos em 91% a quantidade de ABC a serem inseridos na aplicação udp-ipv6 (ou seja, de 22 para 2) e 83% na aplicação webserver/wget (isto é, de 24 para 4), como mostra a Figura 7. A redução é possível pois ao analisar os sistema como todo reduzimos a quantidade de entradas sensíveis e podemos fazer uma análise cruzada de dados. Dessa forma muitos falso positivos são evitados e o número de ABCs pode ser reduzido.

O número de ABCs que conseguimos evitar é significativo principalmente em relação ao custo enérgico de cada ABC inserido e a quantidade de vezes que são acionados. Para cada ABC é necessário incluir seis instruções assembly [Serebryany et al. 2012]. Se consideramos um dispositivo IoT típico com um microcontrolador Atmel AVR de 8 mA, 3V, 7,38 MHz e 1 instrução/clock teremos um gasto de energia de 3,25 mJ/instrução[10]. Portanto, cada ABC incluído aumentará o consumo da aplicação em 19,5 mJ. Se consideramos que uma aplicação udp-ipv6 envia e recebe uma mensagem por minuto teremos ao final de uma dia um consumo de energia de 618 mJ adicionais na abordagem tradicional versus apenas 56 mJ na Siot. A Tabela 3 mostra o impacto dessa economia em termos práticos. Se considerarmos uma rede com cem dispositivos executando essa aplicação durante uma semana, a economia gerada equivale a um quinto

[10] $Energia = I * V * tempo$ onde I é a corrente, V a tensão e $tempo = instruções/clock$.

Aplicação	Instruções	DCFG Nós	DCFG Arestas	Tempo (s)	Memória (MB)
udp-ipv6 cliente e servidor	42.904	65.218	95.237	33,25	45,40
webserver e wget	42.828	66.154	93.678	27,28	92,53

Tabela 2. Número de instruções, tamanho dos DCFGs (nós e arestas) e a quantidade de tempo e memória gastos na análise estática.

Pilha Alcalina	Economia Energética (kJ)	Pilhas (AA) Economizadas	Economia Monetária (R$)
Uma semana	0,393	0,04	0,20
Um mês	1,685	0,16	0,80
Um ano	20,223	1,86	9,27

Tabela 3. Economia de energia.

da capacidade de uma pilha alcalina[11], ou seja, 0,04 kJ ou R$ 1,00. Já em um ano, a economia chega 1,86 pilhas ou R$9,27.

Por fim, é importante frisar que as análises estáticas realizadas pela Siot gastaram em média 29,5 segundos em um notebook Intel core i7 2.20GHz (Tabela 2). Cada aplicação analisada possui mais de 49.000 instruções e cada grafo contém cerca de 65 mil nós e 90 mil arestas. Foram gastos 45,4 MBytes de memória *RAM* na análise da aplicação udp-ipv6 e 92,53 MBytes na análise da aplicação wget/webserver.

6. Trabalhos Correlatos

A maioria das propostas de defesa para sistemas computacionais de maneira geral são baseadas na Análise Estática [Chess and West 2007], na Análise Dinâmica [Serebryany et al. 2012], ou na Análise Híbrida. Em redes tradicionais, as técnicas usadas variam desde análise estática e instrumentação dinâmica de código até soluções que procuram executar o sistema simbolicamente [Chess and West 2007]. No entanto, tais propostas não levam em consideração as características de IoT e, portanto, nem sempre lhes são adequadas [Wangham et al. 2013].

O exemplo canônico é a a proposta de Serebryany *et al.* [Serebryany et al. 2012], ou seja, a ferramenta AddressSanitizer. Essa ferramenta realiza uma análise estática do sistema em busca de acessos ilegais à memória. O sistema é, então, instrumentado para que sua execução seja abortada na iminência de tais acessos. Desta forma, a AddressSanitizer é capaz de eliminar por completo tais vulnerabilidades e, por conseguinte, é chamada de consistente (*sound*). A AddressSanitizer, contudo, é conservadora na sua análise e considera que toda troca de mensagens é contaminada. Tal conservadorismo traduz-se em ineficiência o que a torna, portanto, inadequada para a IoT.

Entre as iniciativas de aplicação das técnicas de análise estática e dinâmica em aplicações que acessam a rede podemos destacar aquelas voltadas para aplicações Web que visam detectar fluxos contaminado e a fim de saneá-los [Huang et al. 2004, Jovanovic et al. 2006, Balzarotti et al. 2008]. Esses trabalhos buscam dependências entre entradas

[11] Custo da pilha R$5,00 e energia por pilha $3Wh$.

(fontes) e saídas sensíveis (sorvedouros), além de verificar se esses caminhos estão saneados, ou seja, se há algum tipo de verificação de vulnerabilidade ou transformação de código que impeça que *strings* enviadas pelos usuários através das entradas do sistema alcancem canais de saída. Mesmo nesses trabalhos não há um tratamento especial em relação às funções que fazem acesso à rede. Cada programa é analisado individualmente e qualquer informação recebida da rede é considerada contaminada. Através das técnicas que apresentamos nesse artigo é possível reduzir o número de entradas (fontes) e, por consequência, reduzir a quantidade de caminhos a serem saneados. Dessa forma é possível reduzir o custo de proteger tais sistemas.

Cumpre lembrar que há propostas para redes limitadas que analisam sistemas. Dentre essas, destacam-se os trabalhos de Sasnauskas *et al.* [Sasnauskas et al. 2010] e Li *et al.* [Li and Regehr 2010], respectivamente, o Kleenet e o T-Check. Tais ferramentas executam sistemas simbolicamente em busca de defeitos de software em geral (*bugs*). A complexidade dos algoritmos dessas soluções comprometem a escalabilidade e, por restrições computacionais, tornam-nas inconsistentes (unsound) já que não possível testar todos os cenários de execução possíveis de um programa.

Nossa solução utiliza o cruzamento de dados de módulos de sistemas distribuídos para ser mais eficiente. Nesta linha, há trabalhos como o de Yang *et al.* [Yang et al. 2009] – sobre modelos de verificação (*model checking*) – e o de Comparetti *et al.* [Comparetti et al. 2009] – sobre engenharia reversa de protocolos. Embora ambas as propostas objetivem a descoberta de vulnerabilidades de protocolos distribuídos, em suas análises, elas tratam os interlocutores da comunicação como caixas-pretas, sem no entanto examinar seus código fonte.

7. Conclusões

A IoT requer soluções de segurança feitas exclusivamente para o seu contexto. É fato que há propostas de segurança para IoT que buscam proteger a IoT contra ataques disparados por adversários externos. Contudo, o mesmo nem sempre ocorre para ataques disparados por adversários com acesso às entradas de dados dos sistema. As propostas para se combater *exploits* foram concebidas no contexto de redes de computadores tradicionais, não consideram as peculiaridades de IoT e, portanto, não lhes são plenamente adequadas. O objetivo deste trabalho foi, portanto, apresentar uma solução de segurança exclusivamente concebida para segurança de código de IoT ante ataques que visam explorar falhas no código como acontece nos ataques de Buffer Overflow.

Nossa proposta – Siot – emprega uma abordagem pioneira na busca de vulnerabilidades. Tal abordagem gera um Grafo de Controle de Fluxo Distribuído que provê ao analisador estático uma visão holística do programa sob análise. Desta forma, é possível cruzar dados de diferentes módulos de um sistema o que torna a solução menos conservadora e, por sua vez, mais eficiente. Os resultados indicam que a Siot provê as mesmas garantias que mecanismos de segurança tradicionais, porém é cerca de 83% mais eficiente em relação a quantidade de verificações de limites de arranjos.

Como trabalhos futuros pretendemos utilizar as técnicas aqui propostas para verificar outros tipos de vulnerabilidades de código como Integer Overfow, por exemplo.

Referências

Atzori, L., Iera, A., and Morabito, G. (2010). The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805.

Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. (2008). Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, pages 387–401. IEEE.

Chess, B. and West, J. (2007). *Secure Programming with Static Analysis*. Addison-Wesley Professional, first edition.

Comparetti, P. M., Wondracek, G., Kruegel, C., and Kirda, E. (2009). Prospex: Protocol specification extraction. In *IEEE Symposium on Security and Privacy*, pages 110–125.

Dunkels, A., Gronvall, B., and Voigt, T. (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors. In *IEEE International Conference on Local Computer Networks (LCN'04)*, pages 455–462.

Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D. T., and Kuo, S.-Y. (2004). Securing web application code by static analysis and runtime protection. In *WWW*, pages 40–51.

Jovanovic, N., Kruegel, C., and Kirda, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Symposium on Security and Privacy*, pages 258–263. IEEE.

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*.

Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.

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

Li, P. and Regehr, J. (2010). T-check: bug finding for sensor networks. In *9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 174–185. ACM.

Oliveira, S., Wong, H. C., and Nogueira, J. M. S. (2006). Nekap: Estabelecimento de chaves resiliente a intrusos em RSSF. In *SBRC - 24th Brazilian Symposium on Computer Networks*.

Sasnauskas, R., Landsiedel, O., Alizai, M. H., Weise, C., Kowalewski, S., and Wehrle, K. (2010). 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*, pages 186–196. ACM.

Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. (2012). Addresssanitizer: a fast address sanity checker. In *USENIX*, pages 28–28. USENIX Association.

Simplício, Jr., M. A., Barreto, P. S. L. M., Margi, C. B., and Carvalho, T. C. M. B. (2010). A survey on key management mechanisms for distributed wireless sensor networks. *Computer Networks*, 54(15):2591–2612.

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.

Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., and Zhou, L. (2009). MODIST: Transparent model checking of unmodified distributed systems. In *USENIX Symposium on Networked Systems Design and Implementation NSDI'09*, pages 213–228.