静的解析技術を用いたIoTシステム検証作業の効率化

堀　武司，本間　稔規

Improvement of IoT System Verification using Static Code Analyzer

Takeshi HORI，Toshinori HONMA

抄　録

IoTシステム利用の広がりに伴い，IoTソフトウェアに含まれるセキュリティ脆弱性の検出を効率的に行う技術が求められている。そこで，脆弱性の発生原因となる不正なメモリ操作などの欠陥を検出する，無償で利用可能なオープンソース静的解析ツールinfer，clang static analyzerの２ツールについて，欠陥検出能力を評価した。組込みTCP/IPスタックTINETに対して両ツールを適用した結果，欠陥検出の偽陽性は少なかったが，既知脆弱性に対する検出率は約15%と低く，検出漏れがある事が判明した。また，IoTソフトウェア環境向けにinferの解析機能を強化する方法として，μITRON OSのサービスコール検証機能を試作し，その性能を評価した。これらの結果にもとづき，従来のソフトウェアテスト作業の一部を静的解析ツールで置き換えて自動化する検証作業の方針を，作業ノウハウ等とともに手引きとしてまとめた。

キーワード：静的解析，セキュリティ脆弱性，infer，clang

1. はじめに

近年，IoT技術が社会に幅広く普及し社会インフラの一部として活用されている。それに伴い，これらを支えるIoTソフトウェアの信頼性やセキュリティ品質の確保が重要な課題となっている。我が国においても，2010年頃から経済産業省が「制御システムセキュリティ」を重要施策の一つとして位置づけており，IoT機器開発者向けのガイド文書[1]を刊行するなどセキュリティ品質確保を呼びかけている。

IoTシステムにはその特性上，(1) 製品開発者が想定しない形でシステム間の通信や相互作用が発生する，(2) 適切に保守管理されない状態の機器が長期間に渡って利用される場合がある，(3) 家電，自動車等の機器では身体や財産に物理的危害を及ぼす可能性がある，(4) 問題が発生してもユーザが気付きにくい，といった特有のセキュリティリスクが存在し，一般の情報システム以上にセキュリティ品質の向上が求められる。そのためIoT機器メーカに対しては製品のセキュリティ脆弱性を確実に除去することが求められ，検証作業の負担が増大している。

ソフトウェアのセキュリティ脆弱性は様々な原因で発生しうるが，米国の情報セキュリティ関連団体MITREの調査[2]では，その原因の半数近くはプログラム内で確保したメモリ領域の境界外での読み込み・書き込み，解放済みメモリ領域へのアクセス，演算結果の桁溢れなどといった，単純なソフトウェア欠陥（バグ）によって占められている。この種の欠陥を効率的に除去しセキュリティ脆弱性を未然に防止することが，IoTシステムにおけるセキュリティ品質向上のための重要な課題の一つである。

近年，ソフトウェア検証技術の分野では，プログラムのソースコードを対象として数理的技法を応用したツールによって機械的に検査し，そこに含まれる欠陥を自動的に検出する静的解析技術が注目されている。静的解析ツールはソースコードのみを検査対象とするため発見できる欠陥の種別には制約があるが，セキュリティ脆弱性の原因となる境界外メモリアクセス等の欠陥検出に適しており，かつ検査作業がツールにより自動化されているため作業工数の削減が可能である。

事業名：経常研究
課題名：静的解析技術を用いたIoTシステム検証作業の効率化に関する研究（令和４～５年度）

産業界でも，自動車や航空宇宙など特に高信頼性が求められる分野では静的解析ツールの普及が進んでおり様々な商用製品が活用されている。しかし，これらの商用ツールはいずれも非常に高価（年額で数百万円）であり，中小ソフトウェア企業が導入することは困難である。一方で，大学等での研究成果を基盤としてオープンソースソフトウェアとして公開されている静的解析ツールもいくつか存在している。そこで本研究では，無償で利用可能なオープンソース静的解析ツールを導入，活用することで，中小ソフトウェア企業におけるIoTシステム検証作業を効率化するための方策について検討した。

2. 無償の静的解析ツールの調査と選定

オープンソースソフトウェアとして無償配布されている静的解析ツールには様々な製品が存在し，機能や性能もそれぞれ異なっている。最初に，本研究で調査対象とする静的解析ツールの候補を次に示す基準に沿って選定した。

1）C言語プログラムの解析に対応すること。

組込みIoTシステム開発ではC言語が用いられる場合が多い。またC言語はポインタを用いたメモリ操作の自由度が高い一方，危険な処理も記述できるため静的解析での検証が有用である。

2）解析作業に人手を介さない，完全な自動解析型ツールであること。

静的解析ツールの一部には対話的な検証作業を伴うタイプ（Frama-C，Verifastなど）もあり検証能力は高いが，人間の作業工数が発生するため効率化には適さない。

3）ツール自体の開発・保守が継続していること。

オープンソースソフトウェアは少数のボランティアで開発される製品も多く，開発・保守活動が中断してしまうリスクがある。そのため開発元の組織体制が安定している製品を選択する。

上記基準に従って検討した結果，infer，及びclang static analyzerの二つのツールを調査対象として選定した。

2.1 infer

infer[3] は，Meta社を中心に開発が進められている静的解析ツールであり，University College London等の研究者が設立したスタートアップ企業Monoidics社で開発された後Facebook社（現Meta社）に買収され，2015年からオープンソースソフトウェアとして公開されている。Java，C，C++，及びObjective-C言語で書かれたプログラムの解析に対応し，境界外メモリアクセス，メモリ等のリソース解放忘れ，並行実行における競合状態など，様々な種類の欠陥検出機能を備えている。

inferの特長は，プログラムの正しさに関する形式体系であるホーア論理を拡張した分離論理[4] にもとづく解析により，ポインタ参照や動的メモリ処理に関する高い解析能力を有する点である。

inferのソースコードはオープンソースライセンスの一種であるMITライセンスで配布されており，商用利用も含め無償利用可能である。

2.2 clang static analyzer

clang static analyzer[5] は，ソフトウェア業界で広く普及しているオープンソースのコンパイラ開発基盤のLLVM及びC/C++/Objective-C言語のコンパイラclangの一部として開発されている静的解析ツールである。clang本体と同じくC，C++，及びObjective-C言語で書かれたプログラムに対する静的解析をサポートする。

clang static analyzerを含むLLVMプロジェクトの成果物はApache License 2.0で頒布されており，inferと同様に商用利用も含めて無償で利用可能である。

3. IoTソフトウェアへの適用による性能評価

オープンソース静的解析ツールの活用を検討するためには，これらのツールがIoTソフトウェアの検証作業においてどの程度の能力を有しているかを把握する必要がある。そこで，実際のオープンソースIoTソフトウェア製品に対してinfer，clang static analyzerの両ツールを適用し，その結果を分析することでツールの解析能力，特に脆弱性につながる欠陥の検出能力の評価を行った。

3.1 適用対象ソフトウェアTINET

ツール適用対象のIoTソフトウェアとして，TOPPERSプロジェクトから配布されているオープンソースのTCP/IPプロトコルスタック実装TINET[6] を選定した。

TINETはμITRON TCP/IP API仕様に準拠したコンパクトなTCP/IP実装であり，組込み向けリアルタイムOS μITRONの一種であるTOPPERS/ASPカーネル上で動作する。TINETは2004年からオープンソースライセンス（TOPPERSライセンス及びBSDライセンス）で公開されており，国内の商用製品を含むIoTシステムでの利用実績がある。そのため，TINETのセキュリティ品質確保は重要な課題であり，TOPPERSプロジェクトでは2020年頃からTINETに含まれるセキュリティ脆弱性の発見と修正に関する活動を組織的に進めてきた。2024年時点の最新版であるTINET1.7では，利用者からの報告，ファジングテスト（大量のランダム入力データを与えて潜在的欠陥を発見する手法）等を用いた検査によって，メモリ境界違反などの脆弱性原因となりうる欠陥が27件発見されており，それぞれコード

の修正も行われている。

3.2 TINETへの静的解析ツール適用試験の結果

TINET1.7のソースコードに対してinfer及びclang static analyzerを適用した。両ツールから出力された欠陥レポートを，TINETソースコードの部位及び欠陥の種類別に集計した結果を表1に示す。

inferの欠陥レポートでは9種類，合計で90件の欠陥が検出された。欠陥の種別では，整数演算の桁溢れ，メモリ境界違反，NULLポインタ参照など多様な種類の欠陥が検出された。inferが得意とするメモリ解放漏れは未検出であるが，これはTINETのコードでは動的メモリ管理がほとんど用いられていないためである。また，欠陥の検出箇所は，TINETの中でもコードの複雑度が比較的高いnetinet6及びtcpモジュールにやや多く分布する傾向が見られた。

clang static analyzerの欠陥レポートでは，NULLポインタ参照2件，未使用代入5件の2種類，合計7件検出され，inferと比べてかなり少ない検出件数となった。また，inferで検出された整数桁溢れ，メモリ境界違反に関してはclangでは検出されなかった。整数桁溢れの未検出は，clangの解析機能の項目に含まれていないためと思われる。しかし，メモリ境界違反の検出についてはclangも対応しており，今回の試験で一件も検出されていない点には疑問が残る。ツール適用方法に不備があった可能性もあるが，詳細な原因は不明である。

3.3 偽陽性の分析

静的解析ツールは，実際には欠陥がない箇所を欠陥として検出（偽陽性）する場合がある。欠陥検出レポートに占める偽陽性の割合が極端に多い場合，人手による確認作業の工数が増大し効率的な検証作業が困難となる。そこで，両ツールの検出結果に関して指摘箇所のコードレビューを行い，検出結果の妥当性を分析した。

inferの総検出項目は90件と多く全箇所のレビューは作業工数上困難であったため，特に重要な欠陥種別と考えられセキュリティ脆弱性と関連が深いメモリ境界違反，NULLポインタ参照，及び未使用代入の3種類（計9件）を調査対象とした。人手によるコードレビューを行った結果，inferが検出した9件すべてで欠陥レポートの内容に合致するコードの不備を発見した。

clang static analyzerによる検出項目は合計7件であるため，同様の確認作業を全件実施した。こちらについても，すべての検出箇所についてコードの不備が発見された。

これらの結果から，対象とする欠陥種別をセキュリティ上重要なものに絞り込んだ場合は偽陽性の影響はさほど大きくなく，inferによる検出結果は検証作業において有効に利用できると考えられる。

3.4 既知脆弱性の検出

3.1節で述べたとおり，TINET1.7にはこれまでに27件の脆弱性が発見されている。今回の試験では脆弱性が修正される前のソースコードを用いているため，静的解析ツールによってこれらの既知脆弱性が発見されることが期待される。そこで既知脆弱性の情報と静的解析ツールの欠陥レポートの内容を比較し，ツールの欠陥検出能力の評価を行った。具体的な評価は次の手順で実施した。

① TINETの脆弱性報告データベースに含まれる項目，及びそれに関連するソースコード修正履歴の情報から，脆弱性が存在する場所と内容を確認する。

表1 TINET1.7に対する静的解析ツールの適用結果

モジュール名	問題点の種別									
	整数桁溢れ	メモリ境界違反	Null ポインタ参照	事前条件違反	未使用代入	Bad footprint	Cannot star	アサーション違反	Abduction case	計
Infer										
net	1	1	-	1	1	2	-	1	1	8
netdev	2	-	-	-	-	-	-	-	-	2
netinet	5	1	-	3	-	2	-	3	-	14
netinet6	9	1	-	9		10	2	3	4	38
tcp	14	-	1	3	4	1	4	-	1	28
計	31	3	1	16	5	15	6	7	6	90
Clang Static Analyzer										
netdev	-	-	-	-	1	-	-	-	-	1
netinet	-	-	1	-	-	-	-	-	-	1
tcp	-	-	1	-	4	-	-	-	-	5
計			2		5					7

h7ga40 commented on Oct 10, 2021

IPv4のARPアドレスにIPv6サイズでコピーしている
Fuzzingにより検出

```
/* Ethernet ARP ヘッダを設定する。*/
memcpy(et_arph->thost, et_arph->shost, ETHER_ADDR_LEN);
memcpy(et_arph->shost, ifaddr->lladdr, ETHER_ADDR_LEN);
- 変更前 memcpy(et_arph->tproto, (uint8_t*)&et_arph->sproto, sizeof(T_IN_ADDR));
+ 変更後 memcpy(et_arph->tproto, (uint8_t*)&et_arph->sproto, IPV4_ADDR_LEN);
ahtonl(et_arph->sproto, taddr);
```

図１　検出に成功したTINET既知脆弱性の例

```
230
231     /* Ethernet ARP ヘッダを設定する。*/
232     memcpy(et_arph->thost, et_arph->shost, ETHER_ADDR_
233     memcpy(et_arph->shost, ifaddr->lladdr, ETHER_ADDR_
        memcpy(et_arph->tproto, (uint8_t*)&et_arph->sproto
234     BUFFER_OVERRUN_L3 Offset added: 16 Size: [0, +oo]
235     ahtonl(et_arph->sproto, taddr); I_13
236
237     /* Ethernet ARP ヘッダを設定する。*/
238     arph->opcode = htons(ARPOP_REPLY); I_12
239
240     /* Ethernet ヘッダを設定する。*/
```

図２　inferによる検出結果

② ツールが出力した欠陥レポートを探索し，ソースコード上の検出位置が近い項目を抽出する。
③ 両者の内容を比較し，ツールが検出した欠陥と既知脆弱性との関連の有無を評価する。

なお，この評価は3.2節の適用試験において，より多数の欠陥を検出したinferの欠陥検出結果を対象として実施した。

評価の結果，既知脆弱性とinferの欠陥レポートの間に直接的な対応関係が認められ検出成功と判断できた事例は，27件中４件（約15%）であった。検出成功と判断された既知脆弱性，及び対応するinferの検出結果の例をそれぞれ図１，２に示す。

図１の箇所ではARP（Address Resolution Protocol）通信の応答パケット構築処理を実装しており，IPv4アドレス（４オクテット長）をメモリ間でコピーする際に誤ってIPv6アドレスのメモリサイズ（16オクテット長）を指定しているため，コピー先で境界外メモリへの書き込みが発生する。これに対してinferは，境界外メモリ領域へのアクセスを示す“BUFFER_OVERRUN_L3”を報告しており，この欠陥を適切に検出できている。

検出できなかった残り23件の既知脆弱性も，その大半はメモリ境界違反に関するものである。入力値等の条件によって欠陥が顕在化するなど図１の事例より複雑であるが，静的解析による検出が期待できる事例である。

静的解析ツールは対象のすべての欠陥検出を保証するものではないが，既知脆弱性に対する検出率15%は高い検出性能とは言えず，inferによる静的解析結果にはかなりの比率で見落としがあることを確認した。

inferの検出性能が十分に発揮できていない原因の一つとして今回対象としたTINETのコードの特殊性が挙げられる。TINETでの通信プロトコルの実装では，メモリ上に展開された通信パケットのデータ列を扱うために，通常のプログラムでは推奨されない低水準の危険なメモリ操作を多用している。例えば，パケットを格納するメモリ領域中のEthernetヘッダやIPヘッダの先頭アドレスを取り出し，ヘッダのデータ構造を表す構造体へのポインタに型変換してアクセスする処理が頻繁に用いられている。このようなスタイルの小さなテストコードを作成して試験したところ，inferのメモリ境界違反の検査が正常に機能しない場合があった。

これ以外にも構造体内部のメンバに関するメモリ境界検査，大域変数が関係する処理など，いくつかのパターンでinferが検出失敗することを確認した。

4. IoTシステム向けのツール機能の拡張

infer等のオープンソース静的解析ツールの多くは，Unix，Windows等の汎用OSや，近年市場拡大が著しいスマートフォン向けソフトウェア環境（iOS，Android等）向けに書かれたプログラムの解析を前提に開発されている。これらの環境向けの静的解析では，プログラム自体に含まれる欠陥の検査に加えて，OS機能を利用するためのサービスコールや標準ライブラリの呼び出しの妥当性に関しても検査が行われる。

一方，IoTシステムでは小型化，省電力性，及びコストなどの面から一般的に用いられる汎用OSではなく，特殊な組込み向けOS，ミドルウェア，ライブラリ等が用いられる場合が多い。この場合でもプログラム自体に含まれる欠陥の検査は可能であるが，対象ソフトウェア環境に固有のライブラリやOSサービスコールに関する情報をツール側で認識していないため，それらに関連する検査は行われない。

組込みIoTシステムのソフトウェア環境は，対象ハードウェアや製品開発プロジェクトに応じて極めて多様な製品が用いられているため，ツール開発元による個別対応は期待できない。

そこで，組込みIoTソフトウェア環境向けに静的解析ツールの機能強化を行う方法について検討した。具体的には，静的解析ツールinferを対象として，国内の組込みシステム開発で比較的高いシェアを有するμITRON OSのサービスコールの解析機能を独自に追加するための試作を行った。

4.1　inferツール実装の調査と機能拡張方法の検討

inferには，第三者による様々な機能拡張手段が提供されている。inferの各種解析機能は個別にモジュール化されており，ユーザ自身で独自の解析機能を開発しツールに組み込

むことも可能であるが，infer本体の実装に用いられる関数型言語OCamlの知識が必要となるなど技術的難易度が高く，一般のツール利用者が独自に実施することは困難である。そのため，より簡便な方法として，OSサービスコール等の振る舞いを解析対象のプログラム言語を用いてモデル化する方法を用いた。

inferはUnix環境向けC言語プログラムに関して，次のような要素を考慮した解析に対応している。

- 標準Cライブラリ
- ヒープメモリの確保，解放（malloc，free等）
- 主要なUNIXサービスコール
- pthread API

これらの解析機能は，inferツールの内部に直接実装するのではなく，解析対象プログラムと同じC言語で記述されたモデルを用意することで実現されている。

図3は，標準Cライブラリに含まれるmemcpy関数のモデル記述を抜き出したものである。memcpy関数の振る舞いは，メモリ領域s2からs1へ長さnのデータをコピーするものであるが，このモデルはプログラムとしてのデータコピー操作を記述したものではない。その代わりに，memcpy関数を呼び出す側の解析対象のプログラムが守るべき条件を仕様として記述している。

- ポインタs1，s2が，正しく割り当てられたメモリ領域を指していること
- 長さnが，s1，s2が指すメモリ領域の長さを超えないこと

モデル中に出現している__required_allocated_array，INFER_EXTRUDE_CONDITION等はinferが認識する特殊関数であり，解析エンジンとの間の情報の受け渡しに用いられる。

```
void* memcpy(void* s1, const void* s2, size_t n) {
  int size_s1;
  int size_s2;
  __require_allocated_array(s1);
  size_s1 = __get_array_length(s1);
  __require_allocated_array(s2);
  size_s2 = __get_array_length(s2);
  INFER_EXCLUDE_CONDITION((n < 0) || (n > size_s1) ||
(n > size_s2));
  return s1;  }
```

図3　inferに含まれるmemcpy（）関数のモデル

4.2　μITRONサービスコール検証機能の試作

infer標準のモデル記述と同様の方法で，μITRON系の組込みOSであるTOPPERS/ASPカーネルのサービスコール呼び出し検証用モデルの試作を行った。

事例としてloc_cpu()，unl_cpu() サービスコールのモデル化について説明する。これらのサービスコールはシステム状態の一つであるCPUロック状態への遷移及び解除を行う。システムがCPUロック状態である期間は一部のサービスコールの呼び出しが制限される。例えば図4ではCPUロック状態でディスパッチ禁止 dis_dsp()を利用しているが，これは不正な呼び出しであり，静的解析等で事前に検出し修正するのが望ましい。

```
void task1() {
  loc_cpu();   // CPU ロック状態に遷移
  dis_dsp();   // ディスパッチ禁止（不正呼び出し）
  unl_cpu(); } // CPU ロック状態の解除
```

図4　CPUロック状態からの不正サービスコール呼出し

```
typedef struct {
  int loc_cpu;
} itron_kernel_state_t;
static itron_kernel_state_t kernel = { 0 };

ER __unl_cpu(itron_kernel_state_t* s){
  s->loc_cpu = 0;
  return E_OK; }
#define unl_cpu() __unl_cpu(&kernel)

ER __loc_cpu(itron_kernel_state_t *s){
  s->loc_cpu = 1;
  return E_OK; }
#define loc_cpu() __loc_cpu(&kernel)

ER__dis_dsp(itron_kernel_state_t* s){
  INFER_EXCLUDE_CONDITION_MSG
  (((s)->loc_cpu==1),
          "dis_dsp() is called on cpu-lock state.");
  ER ret = __infer_nondet_int();
  if (ret) {
    __infer_itron_ena_dsp_state = 0;
    return E_OK;   }   }
```

図5　loc_cpuサービスコールのモデル化

このような「ある状態において，何らかの操作を禁止する」というパターンの記述はinfer標準モデルでも用いられており，それらを参考として図5のモデル化を行った。

システム状態を表現するためのデータ型である__infer_itron_kernel_state_tと，これを保持するグローバル変数を定義した。loc_cpu，unl_cpuのモデル内では，それぞれCPUロック状態を示すフラグの設定，解除の処理を記述した。CPUロック状態での呼び出しが禁止されるdis_dspの内部では，システム状態フラグを検査しCPUロック状態であればinferの特殊関数を用いて欠陥レポートを出力する。なお，最初の試作ではloc_cpu等の内部でシステム状態を保持するグローバル変数の状態を直接書き換えるモデル記述をしていたが，inferの検証機能が正しく機能しなかったため，サービスコールの引数として変数への参照を明示的に渡す形に変更している。実際のサービスコールにはこの引数は存在しないため，C言語のマクロ定義を用いて引数を隠蔽し，本来のμITRONサービスコールの表記と一致させている。

作成したモデルを用いて図４のサンプルプログラムの検証を行った結果を図６に示す。検出箇所の行番号にずれが生じるなどの細かな不備があるものの，CPUロック状態からのdis_dspサービスコール呼出しを不正操作として検出できていることがわかる。

この方法を用いることで，μITRON以外の様々な組込みOS，例えばFreeRTOSなどへの対応も実現できる。また，各種ミドルウェア，ライブラリ等についても，ツール利用者自身によるモデル化が比較的容易に可能となる。

```
Found 1 source file to analyze in
/home/hori/Projects/sa-samples/model/
Test2.c:94 error: Dis Dsp is called on cpu-lock state.
  Detected at line 89、 column 1.
  91.  loc_cpu(); // CPU ロック状態に遷移
  92.  dis_dsp(); // ディスパッチ禁止（不正呼び出し）
  93.  unl_cpu(); // CPU ロック状態を解除
  94. }
```

図６　不正なサービスコール呼び出しの検出例

5. IoTシステム検証効率化の活用方策

２～４章で得られた知見や成果物を活用して，中小企業が行うIoTシステム開発の検証作業にオープンソース静的解析ツールを導入し効率化を図るための方策を検討した。

5.1　静的解析ツールによるテスト作業の置き換え

静的解析の作業はツールによって自動的に実行されるため，ツール適用の準備ができた後は，人が関与する作業工数はほぼゼロである。従来のテスト工程の一部を静的解析で置き換えることができれば大幅な工数削減となるため，具体的なテスト作業の置き換え方針について検討した。

ソフトウェアテストには様々な技法が存在しているが，大まかに分類すると，仕様ベーステストと構造ベーステストに分けられる[7]。

仕様ベーステストは，テスト対象の仕様にもとづいてテスト設計を行い，仕様に対して実装が合致していることを確認する。テスト対象プログラムの内部構造は考慮せずブラックボックスとして扱う場合が多い。

静的解析ツールは原則としてプログラムソースコードのみを対象として解析を行い，仕様の情報は扱わない。そのため，仕様ベーステストを代替する目的では利用できない。

構造ベーステストは，テスト対象のプログラムの内部構造に着目し，構造要素に関して漏れなくテストすることを目標としてテスト設計を行う。プログラムの実行フローの構造に着目する場合は，文網羅，分岐網羅などの基準にもとづいて，高い網羅率（カバレッジ）を目標としてテストケースを設計し実行する。この場合のテスト目的は，様々な実行条件の下でプログラムが正しく動作することと考えられるが，その中には，仕様との合致だけではなく，メモリ境界違反やNULL参照などの，プログラムとしての明らかな欠陥がないことの確認も含まれる。この部分は静的解析ツールの機能と共通であるため，ツールによる置き換え，自動化の対象となる。

静的解析ツールは原理上100%の欠陥検出を保証するものではないが，従来の構造ベーステストでも完全な欠陥検出ができない点は同じである。静的解析ツールの特性を把握し検出性能が十分に引き出された状態であれば，少数の有限なテストケースでテストを行うよりも静的解析の方が実質的な網羅度が高くなることも考えられる。

5.2　技術導入のための手引き

infer等のオープンソース静的解析ツールは大規模，複雑なソフトウェアであり，機能を理解し，使いこなすことは容易ではない。また，関連する技術情報等も不足しており，特に日本語で系統的にまとめられた情報は少ない。

そこで，道内中小企業への技術支援における活用を想定して，静的解析導入による検証作業効率化の方針，ツール活用のための技術ノウハウ等を手引きとして取りまとめた。手引きでは図７に示す作業効率化方針を提案し，その概要は以下のとおりである。

① 設計・実装工程
静的解析ツールが苦手とするパターンを把握し，解析しやすい形でプログラム設計を行う。危険なメモリ・ポインタ操作を最小限とする。関数，モジュールの規模を小さく保ち，複雑度を下げる。大域変数の利用を減らす。

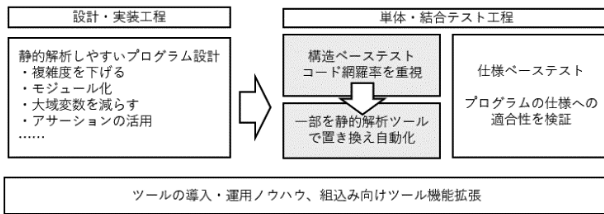

図７　提案する検証作業効率化方針の概要

② 単体・結合テスト工程
各種テスト作業のうち，構造ベーステストの一部を静的解析ツールで置き換え，自動化によりテスト工数の削減を図る。

③ ツールの導入・運用に関するノウハウ
ツールのインストール手順。IoTプロジェクトへのツール適用手順。欠陥レポートの解釈。特殊なOS及びソフトウェア環境に向けたツール機能の強化。その他。

6. まとめ

IoTソフトウェア脆弱性の発生原因となる不正なメモリ操作などの欠陥を効率的に検出するため，無償で利用可能なオープンソース静的解析ツールinfer及びclang static analyzerの能力評価を行った。組込みTCP/IPスタックTINETに対して両ツールを適用した結果，欠陥検出の偽陽性は少ないことが確認できた。しかし，静的解析ツールが不得手とするプログラムコードを多く含むTINETの既知脆弱性の検出率は約15%と低い結果となり検出漏れがあることが判明した。また，IoTソフトウェア環境向けにinferの解析機能を強化する方法を検討し，組込みμITRON OSサービスコール検証用のモデルを試作した。これらの結果にもとづき，従来のテスト作業の一部を静的解析ツールで置き換え自動化する効率化方針を検討し，作業ノウハウ等とともに手引きとしてまとめた。

今後は，TINET以外のIoTソフトウェアを対象とした欠陥検出能力の評価を実施し，様々なプログラムコードに対する検出の成功，失敗の条件など，ツールの精密な挙動把握を行う。これらの結果にもとづき手引きの内容の改善を進き，企業等への技術支援への活用を予定している。

引用文献

1）IPA/SEC編：つながる世界の開発指針，情報処理推進機構,(2017)
2）MITRE：2020 CWE Top25 Most Dangerous Software Weaknesses, https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html
3）https://www.fbinfer.com/
4）O'Hearn："Separation logic", Communications of the ACM, Volume62, Issue2,(2019)
5）https://clang-analyzer.llvm.org/
6）阿部　司，吉村　斎，他："組込みシステム用TCP/IPプロトコルスタックの実装と評価"，情報処理学会論文誌44(6) 1583-1592,(2003)
7）ISO/IEC 29119-4：2021 Software and systems engineering-Software testing, Part4：Test techniques