修 士 論 文

組込みソフトウェアへの故障注入攻撃に対する安全性評価に関する研究

情報基礎科学専攻
梨本 翔永

目次

第 1 章

緒言

近年，Internet of Things (IoT) という次世代ネットワーク技術に注目が集まっており，従来ネットワークから隔絶されていた組込み機器においても，インターネット接続が進むことが予想されている．様々な機器をインターネットに接続してデータを収集し，解析した結果をもとに機器を制御することで，作業やエネルギーの高効率化，機器制御の自動化が可能になることが期待されている．しかし，IoT の活用に関しては課題もあり，その中でも特に，組込み機器におけるセキュリティ（組込みセキュリティ）が大きな関心を集めている [1], [2]．セキュリティが重要視されている理由は大きく二つある．一つは，攻撃者にとっての攻撃手段が増加することである．インターネット接続された組込み機器に関しては，従来の汎用 PC への攻撃と同様に，遠隔からの攻撃が可能となる．もう一つは，攻撃の成功時に得られる利益が増大することである．例えば，攻撃に成功した組込み機器を利用して，ネットワークを介して接続された他の機器へ攻撃を行うことが考えられる．また，IoT では，センサネットワークのように，同一機種が大量に使用されることが予想されるため，ある機器に対して有効な攻撃は，他の機器においても適用できる可能性が高い．このような理由から，現在，組込みセキュリティの研究が盛んに行われている．その中でも特に，身近な市販品を対象とした攻撃は広く注目を集めている．これまでに，ネットワーク対応のセキュリティカメラ [3] や Wi-Fi 接続する自動車 [4]，SmartTV[5], [6] の制御を奪取可能であることが報告されている．このように，ネットワーク接続する組込み機器への攻撃は現実的な問題となっている．

高度な応用向けに使用されているハイエンドな組込み機器への攻撃に注目が集まる一方で，ローエンドな組込み機器のセキュリティの必要性も高まることが予想される．例えば，センサネットワークのような応用では，多くの機器が相互に接続されて情報の伝達を行う．そのようなシステムを構成するセンサノードは大量に必要となることや，高性能なマイコンが必要とされるような高度な処理は行わないことから，低価格なローエンド向けマイコンが使われる場合が多い [7]．そうした機器では使用可能なリソースが大きく制限

されるため，セキュア OS を搭載していないものや，許可されない領域へのメモリアクセスを禁止する等のセキュリティ機能が提供されていないものも多く存在する．このような，多くの機器の集合によりシステムが構成される応用に関しては，一台の機器への攻撃がシステム全体に影響を及ぼす恐れがある．また，攻撃者はシステムの最も脆弱な部分を攻撃の足掛かりとすることから，こうしたリソース制約の厳しいローエンドな機器におけるセキュリティも疎かにすることはできない．

一方で，そのような組込み機器への攻撃に関しては，ソフトウェアへの論理攻撃の他に，ハードウェアへの物理攻撃も存在する．物理攻撃とは，機器の物理的な情報，あるいは物理的な手段を用いた機器への攻撃のことである．IoT で使用される機器の中には，攻撃者が物理的にアクセス可能なものも存在することから，物理攻撃の可能性を検討する必要がある．例えば，センサネットワークで使用されるノードは，管理者の手元を離れた比較的監視の少ない場所で使用されることがある．また，スマートカードや携帯電話のように，攻撃対象となる機器が攻撃者の手元に存在する場合も多い．このような理由から，組込み機器への物理攻撃に関しても，ソフトウェアへの論理攻撃と同様に，予め，想定される攻撃への対策を行うことが重要だと考えられる．

故障注入攻撃は組込み機器に対する物理攻撃の一つであり，特に，暗号プロセッサへの攻撃手法として知られている．故障注入攻撃では，まず，暗号が動作している機器へ物理的な手段を用いて故障を注入する．こうして注入された故障により，ビット反転や命令スキップといった現象が生じる [8], [9], [10]．そのようにして得られた故障の結果を解析することにより，機器の秘密鍵を取得する．これまでに，RSA や ECC (Elliptic Curve Cryptography) などの公開鍵暗号への攻撃手法 [11], [12] や，DES や AES などの共通鍵暗号への攻撃手法 [13], [14] が報告されている．

一方，近年では，ビット反転や命令スキップといった現象に着目して，マイコン上の汎用ソフトウェアへの攻撃手法として故障注入攻撃が適用可能であることが報告されている．メモリのビット反転を利用した攻撃では，Java Card や汎用 PC 上で動作する Java Virtual Machine で，故障によりバイトコード検証器を無効化することやリターンアドレスを直接書き換えることにより，任意コードを実行可能であること [15], [16], [17] が報告されている．一方で，命令スキップを利用した攻撃では，関数呼び出しの発生時に故障を注入し，バッファオーバーフローと同様の効果を発生させることが可能であること [18] が報告されている．このように，従来の攻撃に対しては有効な対策が施されている場合であっても，故障注入攻撃を用いることで無効化される危険性がある．

上記の汎用ソフトウェアへの故障注入攻撃に関する一連の報告では，単一の故障を利用した攻撃しか報告されていなかった．一方で，近年では，暗号プロセッサへの故障注入手法として複数回の故障を注入する多重故障注入を利用した攻撃が報告されている [19], [20]．こうした報告により，多重故障注入のための効果的な手法が提案されるとともに，

マイコンへ多重に故障を注入した際の影響が明らかになり，そうした攻撃の有効性が認識されてきている．

そこで，本論文では，組込みソフトウェアに対する多重故障注入攻撃を提案し，その実現可能性を明らかにするとともに，そうした攻撃への対策手法を示す．提案攻撃の基本的なアイデアは，マイコンへの故障注入により高頻度で生じる命令スキップに着目し，多重故障注入を利用して任意の命令を複数個スキップすることである．そのような故障注入攻撃と従来の攻撃を併用することにより，従来攻撃に対策済みのソフトウェアであっても，その対策が無効化される可能性があることを実証する．具体的には，従来攻撃の例として，ソフトウェアへの攻撃としてよく知られるバッファオーバーフロー (Buffer overflow: BOF) 攻撃 [21] に着目し，BOF 攻撃と故障注入攻撃を併用した攻撃手法を提案する．BOF 攻撃は外部からの入力データを処理するソフトウェアにおいてしばしば発生し，任意コードの実行が可能になる場合も多いため，非常に危険性の高い攻撃である．IoT で活用されるような機器においては，外部からの入力を受け付けることは必須であるため，BOF 攻撃の標的となる可能性は高い．本提案攻撃の有効性は，(i) 8 ビット AVR マイコン，(ii) 32 ビット ARM マイコンの二種類のマイコンを使用した実験を通して実証する．

(i) の 8 ビット AVR マイコンは，比較的動作周波数が低く，パイプライン構造などのアーキテクチャも単純である．このような 8 ビットマイコンは，ローエンドな機器に求められる用途の中でも，特に，簡単な処理しか行わず，低価格・低消費電力に重点を置く場合に使用される [7]．まずは，そのように構造が単純で比較的容易に攻撃が可能なマイコンを用いて実験を行い，提案攻撃の有効性を実証する．(ii) の 32 ビット ARM マイコンは (i) の AVR マイコンと比べると，動作周波数も高く，アーキテクチャも複雑になっている．ARM マイコンはアーキテクチャの種類も非常に多く，ローエンドな機器からハイエンドな機器まで，様々な用途で使用されている．その中でも特に，8 / 16 ビットマイコンの後継機として期待されている Cortex-M0+ に着目する．Cortex-M0+ は Cortex-M シリーズの中でもエネルギー効率を重視した設計となっている．このような機器への攻撃実験を通し，特定の機器やアーキテクチャに限定された攻撃手法ではなく，幅広い機器に対して有効であることを実証する．

本論文は，以上の内容を取りまとめたものであり，以下に示す 5 章によって構成する．

第 1 章は，本研究の背景と目的および本論文の概説をまとめた緒言である．

第 2 章では，組込みソフトウェアに対する論理攻撃と物理攻撃について述べる．まず，ソフトウェアへの代表的な論理攻撃について述べ，その中でも，特に，BOF 攻撃に着目し，攻撃の原理とその対策手法について述べる．次に，ハードウェアへの物理攻撃の分類を行い，その中でも，特に，故障注入攻撃に着目し，故障の注入手法に基づく分類を行い，故障注入を用いたソフトウェアへの攻撃例について述べる．また，故障注入攻撃への対策

手法についても述べる．

第 3 章では，本論文で提案する，故障注入攻撃を併用したバッファオーバーフロー攻撃について述べる．まず，本論文と多くの類似点を有する文献 [18] の研究に着目し，比較を行うとともに，提案攻撃の概念を示す．次に，AVR マイコンと ARM マイコンを対象とし，攻撃の適用方法を述べる．

第 4 章では，実験を通して提案攻撃の有効性を実証するとともに，対策方法について述べる．まず，AVR マイコンと ARM マイコンのそれぞれに対する攻撃実験を行う．その後，各マイコンについてソフトウェアレベルでの対策を施し，そのオーバーヘッドを評価する．

第 5 章は，結言である．

以上，本論文の企図するところを概説した．

第 2 章

組込みソフトウェアへの攻撃に関する基礎的考察

2.1 はじめに

本章では，組込みソフトウェアへの攻撃に関する基礎的考察を行う．まず，ソフトウェアへの論理攻撃について述べ，特に，バッファオーバーフロー (Buffer Overflow: BOF) 攻撃に着目し，攻撃原理とその対策手法について述べる．次に，ハードウェアへの物理攻撃について述べ，特に，故障注入攻撃に着目し，攻撃手法や既存研究について述べる．また，対策手法についても述べる．

2.2 ソフトウェアへの論理攻撃

本節では，ソフトウェアへの論理攻撃について述べる．まず，攻撃の際に利用される脆弱性に基づいて，ソフトウェアへの代表的な論理攻撃について述べる．次に，特に，BOF 攻撃に着目し，同攻撃の概要と原理について述べる．最後に，従来の BOF 攻撃対策として主要なものについて述べ，ローエンドなマイコン上で実装することを考慮した場合に，有効な手法を選択する．

2.2.1 代表的な論理攻撃

論理攻撃とは，仕様と入出力を用いる攻撃のことを指し [22]，このような，攻撃の原因となる弱点を脆弱性と呼ぶ．ソフトウェアへの攻撃の目的には，主に次のようなことが挙げられる [23].

- 管理者・一般ユーザの権限奪取
- サービス妨害 (Denial of Service: DoS)
- 情報の改ざん
- 機密情報の取得

表 2.1　種類別脆弱性割合 (全体) (文献 [21] の図 6 より)

脆弱性	CWE 番号	割合 [%]
Buffer Errors	119	14
XSS (Cross site scripting)	79	13
Access Control	264	11
SQL Injection	89	10
Input Validation	20	10

表 2.2　種類別脆弱性割合 (最高危険度) (文献 [21] の図 8 より)

脆弱性	CWE 番号	割合 [%]
Buffer Errors	119	35
Not enough info	-	21
Access Control	264	8
Input Validation	20	6
Code Injection	94	5

- 不正プログラムの実行

こうした目的のために，攻撃者はソフトウェアに存在する様々な脆弱性を利用して攻撃を行う．ここでは，攻撃の際に利用する脆弱性を基にして，代表的な論理攻撃について述べる．脆弱性の識別には，共通脆弱性タイプ一覧 CWE (Common Weakness Enumeration)[24] が，脆弱性の危険度評価基準としては，共通脆弱性評価システム CVSS (Common Vulnerability Scoring System)[25] が広く用いられている．このような共通の脆弱性指標を用いることで，各機関で保有する脆弱性データベース間で互換性が保たれることや，同じ指標を用いて議論が可能になることが利点として挙げられる．ソフトウェアの脆弱性は日々報告されており，CWE と共に運用される CVE (Common Vulnerabilities and Exposures) や NIST (National Institute of Standards and Technology) の NVD (National Vulnerability Database)[26]，国内では，JVN (Japan Vulnerability Notes) iPedia[27] が脆弱性情報データベースとしてよく知られている．

文献 [21] において報告された，1988 年〜2012 年の 25 年間に CVE, NVD へ登録された脆弱性の統計情報をまとめたものから上位 5 種類の脆弱性を抜粋したものを表 2.1, 2.2 に示す．それぞれ，CWE 番号と割合を併記した．表 2.1 は全体における割合を示し，表 2.2 は最高危険度における割合を示す．なお，表 2.2 において 2 番目に多い Not enough info は報告された際の情報が不十分で分類不能であった脆弱性をまとめたものである．表 2.1, 2.2 に示す通り，Buffer Errors 脆弱性が，全体の中でも，最高危険度の脆弱性の中でも，最も報告件数が多い．これは，Buffer Errors 脆弱性はアプリケーションを問わず，あらゆるソフトウェアに存在し得ることから，古くから現在に至るまで報告が絶えないためである．Buffer Errors 脆弱性は，設計者が意図したメモリバッファの境界外からの読み込みや境界外への書き込みが可能な場合に存在する．Buffer Errors 脆弱性が存在する場合，BOF 攻撃が成功する危険性が極めて高い．BOF 攻撃により任意コードの実行が可能である場合が多いことから，表 2.2 に示すように，最高危険度の割合においても最も報告件数が多くなっている．

また，表 2.1 で，Buffer Errors 脆弱性に次いで報告件数が多い XSS 脆弱性はユーザからの不正な入力（悪意のあるスクリプト）を元に，動的に Web ページを出力する場合に存在する．攻撃者は，他人の Web ページに埋め込んだスクリプトを被害者に実行させることで，Web ページへのアクセス制御に用いているクッキー情報や個人情報を奪取する．

XSS 脆弱性に次いで多い Access Control 脆弱性はデータやプログラム等のリソースや Web ページへのアクセス権限が適切に設定されていない場合に存在する．Access Control 脆弱性が存在する場合，各種リソースへの不正アクセスが行われ，機密情報が取得される危険性が高い．このような重大な被害をもたらすことから，Access Control 脆弱性は最大危険度の割合においても 3 番目に報告件数が多くなっている．

Access Control 脆弱性の次に報告件数が多い SQL Injection 脆弱性は，リレーショナル データベースを扱うための言語である SQL に依存した脆弱性である．特に，ソフトウェアの中で動的に SQL コマンドを発行しており，かつ，作成した SQL コマンドを攻撃者が改ざん可能である場合に存在する．クエリと呼ばれる SQL コマンド発行の要求を直接送信することや，部分的に不正なコマンドをクエリに組み込むことで，攻撃者は，データベースの内容を改ざんすることや不正に取得することが可能となる．

SQL Injection 脆弱性に次いで多い Input Validation 脆弱性は入力データの確認が適切に行われていない場合に存在する．例えば，入力可能な数値範囲を超えた不正な入力や悪意のあるスクリプトの入力，C 言語におけるデフォルト値の設定がされていない case 文などが挙げられる．表 2.2 においても 4 番目に報告件数が多い脆弱性であるが，これは，不正な入力によりソフトウェアが異常終了することや，不正なコマンドの実行が可能になる場合が多く，被害の程度が大きいためだと考えられる．さらに，アプリケーションの種類に依らず生じることから，報告件数も多くなっていると予測できる．

また，表 2.2 で 5 番目に多い Code Injection 脆弱性は，SQL Injection 脆弱性と同じように，攻撃者の不正な入力がコマンドとして解釈される場合に存在する．shell command に代表される OS コマンドの使用により，直接リソースにアクセス可能であることから，機密情報の取得やサーバ上のプログラムの実行が可能となり得ることから，危険度の高い

脆弱性である．

ローエンドなマイコン上に実装した組込みソフトウェアを想定すると，Web アプリケーションを想定する XSS や SQL Injection の脆弱性が存在する可能性は低いと考えられる．一方，アプリケーションに依存せず存在する Buffer Errors や Access Control, Invalid Validation の脆弱性は，多くの組込みソフトウェアにおいて発生する可能性が高い．その中でも Buffer Errors 脆弱性は，入出力を必須とする IoT アプリケーションにおいても高頻度で存在する可能性があることや，報告件数も多く，攻撃による影響度が大きいことから，特に関心度が高い脆弱性であると言える．そこで，本論文では，ソフトウェアへの論理攻撃として，特に，Buffer Errors 脆弱性と密接に関連する BOF 攻撃に着目する．

2.2.2　バッファオーバーフロー攻撃の原理

BOF は，確保したメモリ領域を超えてデータが入力されることにより発生する不正なメモリの上書きである．BOF は，ソフトウェアの入力データ処理で，設計者の想定を超える入力データを与えた時に発生することが多い．手動でメモリ管理を行い，細かな操作が可能である C/C++ 言語を使用したプログラムにおいて，設計者がメモリを適切に管理していない場合にしばしば，BOF 脆弱性が顕在化する．BOF 脆弱性の原因となり得る標準ライブラリ関数には，次のようなものがある [28], [29].

- `gets()`
- `strcpy()`
- `strcat()`
- `sprintf()`
- `vsprintf()`
- `scanf()`
- `sscanf()`
- `fscanf()`
- `vscanf()`
- `vsscanf()`
- `vfscanf()`

以上の関数を不適切に使用している場合には BOF が発生する可能性が高い．BOF 攻撃とは， BOF を意図的に利用することでプロセッサに不正な処理を実行させる攻撃である．BOF 攻撃では，メモリを攻撃コードで上書きし，任意のコードやプログラム中の任意の関数を実行する．この結果，システムへの侵入や管理者権限の奪取が可能になる可能性が高く，重大な被害をもたらすことが多い．

```
void func(char *arg){
    char buf[20];
    strcpy(buf, arg);
}

void main(int argc, char *argv[]){
    func(argv[1]);
}
```

図 2.1　BOF 脆弱性のあるプログラム

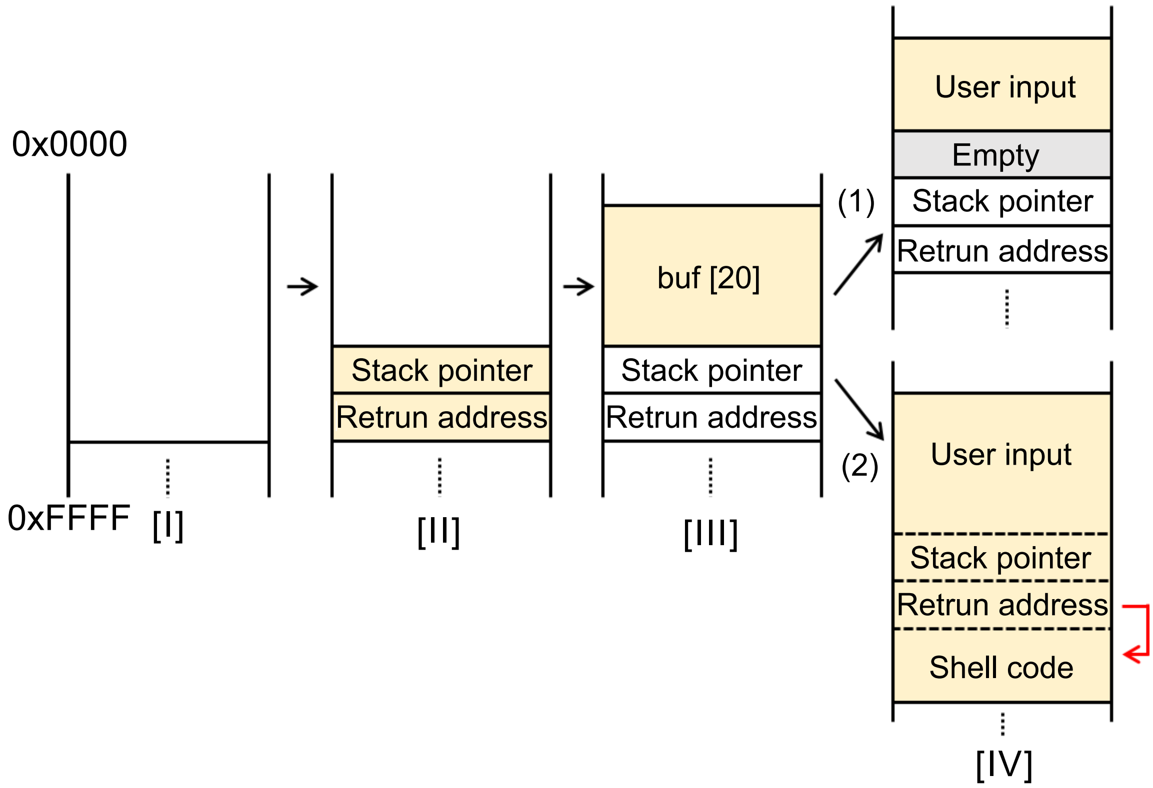


図 2.2　図 2.1 のプログラムにおけるスタック領域の変化

BOF には，スタックメモリ上で発生するスタック BOF と，ヒープメモリ上で発生するヒープ BOF の 2 種類が存在する．スタックメモリは関数呼び出しと密接に関連しており，リターンアドレス等のサブルーチンの情報が格納される．そのため，BOF 攻撃では，ヒープ BOF よりもスタック BOF の方が，より攻撃に用いられやすく，報告件数も多い [29]．そこで，ここでは，スタック BOF の原理を，図 2.1 に示すプログラムと，図 2.2 に示すスタック領域の変化の流れ図を用いて説明する．図 2.2 [I] は，図 2.1 のプログラムにおいてスタートアップルーチンによる初期化処理が終了し，main 関数が呼ばれた時のス

表 2.3　代表的なバッファオーバーフロー攻撃対策手法

手法名	対策方法	レイヤ
メモリ空間のランダム化	スタック配置アドレスをプログラムの実行毎に変更	OS (Operating system)
スタック保護	関数呼び出し時，スタック上に乱数値（カナリアワード）を付与し，リターン時にチェック	コンパイラ
データ実行防止	スタック領域のコードは実行しない	OS, CPU，コンパイラ
入力データのサイズ制限	引数で入力バイト数を指定可能な関数を使用	プログラム

タックの状態を表す．この後，7 行目では，標準入力として与えられた文字列を引数にして，関数 `func()` を適用する．この時，関数呼び出しが発生するため，リターンアドレスとスタックポインタがスタックに保存される (図 2.2 [II])．その後，2 行目で宣言されている `buf` という変数のために 20 バイトの保存用領域がスタックに確保される (図 2.2 [III])．このように，`malloc()`，`calloc()`，`realloc()` といった動的メモリ確保関数を使わずに静的に宣言された変数は，自動的にスタック上にその保存用領域が確保される．次の処理 (3 行目) では，`strcpy()` を使用して，引数である `arg` を，先ほど宣言した `buf` へとコピーする．設計者の意図通りであれば，図 2.2 [IV]-(1) のように，引数として与えられたユーザの標準入力 `argv[1]` はいくらか余裕を残して `buf` へコピーされる．しかし，`strcpy()` はコピー元とコピー先のサイズを考慮していないため，`buf` のサイズを超えてデータを与えることが可能である．こうしてメモリの不正な上書き，すなわち BOF が発生する．BOF 脆弱性を利用することで，図 2.2 [IV]-(2) のように，確保したメモリ領域を超えてデータを上書きすることが可能となる．この時，リターンアドレスを任意のアドレスに書き換えることで，不正な処理を実行することができる．この例では，送信データに含まれる攻撃コード (シェルコード) のあるアドレスを指すようにリターンアドレスを上書きしており，任意のコードを実行させることが可能となっている．

2.2.3　バッファオーバーフロー攻撃への対策

BOF 攻撃は，その被害の大きさから重要視されており，様々な対策技術が開発されている [30]．典型的な対策を表 2.3 に示す．メモリ空間のランダム化 [31] では，プログラムの実行の度にスタックやヒープ等の配置アドレスをランダム化する．これにより，攻撃者は，不正に呼び出そうとする関数のアドレスを推測することが困難となる．スタック保護

[32] では，関数呼び出しの際にカナリアワードと呼ばれる乱数値をスタックに格納し，スタック領域の書き換えを検知する．呼び出した関数の処理が終了した後にカナリアワードを検査し，改ざんされていないことを確認する．カナリアワードの推測が困難であることから，BOF 攻撃対策となる．データ実行防止 [33], [34] では，スタック領域のコードを実行することがそもそも不正な動作であるとし，これを禁止する．入力データのサイズ制限では，入力として受け取るバイト数を指定可能な関数を利用することで BOF の発生を防ぐ．これは，BOF 脆弱性を有する関数の問題点であった，入力サイズを制限していないことに注目している．この対策は，他の 3 種類の対策と比べ，BOF を利用した攻撃だけでなく，BOF の発生自体も防げるという利点がある．

組込み機器では AVR や ARM マイコンが広く使用されている．そのようなマイコンの中でも安価なものに関しては，リソースの制約から OS (Operating System) が搭載されていない場合やスタック保護などの高度な対策が利用できない場合が多い．このため，機種依存性が低く，ローエンドなマイコンからハイエンドなマイコンまで幅広く利用可能な，入力データのサイズ制限が最も一般的な BOF 攻撃対策であると言える．例えば，図 2.1 で使用した `strcpy()` の場合，`strncpy()` を代わりに使用することで，入力データのサイズに上限を設けることが可能である [29], [35].

2.3 ハードウェアへの物理攻撃

本節では，ハードウェアへの物理攻撃について述べる．まず，物理攻撃の分類を行い，その中でも，特に，ソフトウェアへの攻撃に適用可能な故障注入攻撃に着目をする．まず，ソフトウェアに対する故障注入攻撃を行った既存研究において使用されている注入手法をまとめ，それぞれの特徴を述べる．次に，そうした状況を踏まえ，本実験で使用する故障注入手法として適するものを選択する．最後に，故障注入攻撃への対策手法について述べる．

2.3.1 物理攻撃の分類

物理攻撃とは，機器の物理的な情報，あるいは物理的な手段を用いた機器への攻撃のことである．ハードウェアへの物理攻撃は大きく，侵襲型と非侵襲型の二つに分けられる [22]．侵襲型の攻撃では，パッケージを開封して内部の回路を直接操作するため，攻撃者が得られる情報は多く自由度が高い反面，情報の取得や解析を行うためには高度な技術的知識や解析装置が必要であるという欠点がある．一方，非侵襲型の攻撃では，侵襲型の攻撃ほど高い自由度はないものの，パッケージを開封せずに攻撃を実行できるため攻撃の痕跡が残らず，また，比較的安価な装置で実行できるという利点がある．非侵襲型の攻撃

は，さらに，受動型と能動型の二つに分けられる．受動型の攻撃はサイドチャネル攻撃と言われ，機器の漏洩情報（サイドチャネル情報）を利用して攻撃を行う．サイドチャネル攻撃は暗号プロセッサへの攻撃手法であり，暗号処理中の電磁波や計算時間といったサイドチャネル情報を観測することで，秘密鍵を推測する．一方，代表的な能動型攻撃としては故障注入攻撃が挙げられる．故障注入攻撃も広い意味ではサイドチャネル攻撃と呼ばれるが，本論文では，非侵襲型でかつ受動型の攻撃である狭義サイドチャネル攻撃のみをサイドチャネル攻撃と呼ぶ．故障注入攻撃もサイドチャネル攻撃と同様に，暗号プロセッサへの攻撃として知られている．故障注入攻撃では，まず，暗号が動作している機器へ物理的な手段を用いて故障を注入する．注入された故障により生じる命令スキップやビット反転といった影響を解析することで，秘密鍵を推測する．

組込みソフトウェアの制御を奪うことを目的として物理攻撃を用いることを考えた場合，受動的な攻撃方法であるサイドチャネル攻撃は適さない．侵襲型の攻撃を利用する場合，回路を直接改変し，従来の攻撃対策を無効化してソフトウェアへの攻撃を実行することが考えられるが，ローエンドな機器への攻撃を想定した場合，多くの場合，攻撃により得られる利益は攻撃に必要なコストに見合わない．さらに，侵襲型の攻撃は，攻撃者が攻撃対象とする機器の制御を掌握している必要であり，攻撃対象が大きく制限されるという欠点も存在する．一方，非侵襲型の故障注入攻撃は，攻撃に要するコストも比較的少なく [36]，注入した故障により生じる命令スキップやビット反転といった現象がソフトウェアへの攻撃に利用可能である [15], [16], [17], [18] ことから，組込みソフトウェアに対する攻撃として現実的な脅威となり得ることが予想できる．そこで，本論文では，ハードウェアへの物理攻撃として，特に，故障注入攻撃に着目する．

2.3.2　故障注入攻撃の分類

故障注入攻撃で用いる故障には，永久故障と過渡故障の 2 種類が存在する．永久故障では，故障を注入することにより LSI 上のセルに直接変化を起こすため，LSI の機能は永続的に回復することがない．そのため，攻撃のために必要な故障を特定のセルに注入することは非常に難しく，また，機能障害が継続するため故障の検知も比較的容易であることから，攻撃の影響度は小さいと考える．一方，過渡故障では，注入された故障により，一時的にデータや演算処理に誤りが生じる．こうして引き起こされた誤りは，回路のリセットや故障を注入することをやめることで回復する．そのため，攻撃の痕跡は残らず，故障の検知も永久故障と比べて困難である．また，特定の故障が得られるまで繰り返し故障を注入することが可能であるため，故障の影響の解析に適している．以上の理由から，故障注入攻撃に関する多くの研究では過渡故障が用いられており，様々な攻撃手法や故障注入手法が研究されている．そこで，本論文では，特に，過渡故障のみを扱う．

表 2.4　故障注入手法の分類

注入手法	分解能 空間	分解能 時間	コスト
(1) 電源電圧の変化	低い	中	安価
(2) クロック周波数の変化	低い	高い	安価
(3) 温度の変化	低い	低い	安価
(4) 電磁波の照射	低い	中	安価
(5) 白色光の照射	低い	低い	安価
(6) レーザの照射	高い	高い	高価

暗号プロセッサに対する故障注入攻撃では，誤りを含む暗号文から秘密鍵を推測する解析手法と並行して，故障注入手法も多く提案されている．その中から，文献 [36] に基づき，典型的な故障注入手法を表 2.4 にまとめる．あわせて，位置的な分解能と時間的な分解能，金銭的コストを記した．位置的な分解能は「低い・高い」の 2 段階，時間的な分解能は「低い・中・高い」の 3 段階で表す．この基準は定量的な値に基づくものではなく，各故障注入手法を比較した時の程度を表す．また，金銭的なコストに関しては，$3,000 より安い場合には「安価」，$3,000 以上である場合には「高価」となっている．

(1), (2) の手法では，デバイスに供給する電源電圧やクロック周波数を急激に変化させることで，フリップフロップのセットアップタイム違反を発生させる．その結果，命令のフェッチやデコード，あるいはデータの読み書きに誤りが生じる．(1), (2) の手法は欠陥（グリッチ）の入ったリソースを供給するという特徴から，それぞれ，パワーグリッチ，クロックグリッチと呼ばれている．これらの手法は，故障の注入タイミングを細かく決定できることや，必要な装置が安価であることから，多くの研究で使用されている [8], [20], [37], [38].

(3) の手法では，機器を高温な状態にすることにより，(4) の手法では電磁波を照射することにより，それぞれメモリの値を変化させる [15], [39]．これにより，データのビットが反転することや，特定のデータの全ビットが 0 に，あるいは 1 に変化すること，または不定値になることが知られている．これらの手法では，機器に直接手を加える必要がない場合が多く，その他の手法と比べて侵襲性が少ないという利点がある．

(5), (6) の手法では，光の照射により光電効果を引き起こし，直接メモリの値を変化させる [40]．(5) の手法では，カメラのフラッシュや蛍光灯などの安価な装置によって実現可能ではあるが，位置的にも時間的にも分解能が低いという欠点がある．一方，(6) の手法では，位置的にも時間的にも分解能の高いレーザを用いることで，より効果的に攻撃を

表 2.5　既存研究における故障注入手法と故障モデル

文献	故障注入手法	故障モデル
[15]	温度変化	ビット反転
[16]	レーザ照射	命令スキップ
[17]	レーザ照射	データ値変化 (全ビット 1 に)
[18]	クロック周波数の変化	命令スキップ

実行できる反面，装置が高価であるという欠点がある．また，LSI に直接光を照射する必要があるため，パッケージの開封が必須で，その他の手法に比べて高い侵襲性を要するという欠点もある．

2.3.3　ソフトウェアへの故障注入攻撃

従来は，暗号ソフトウェアや暗号ハードウェアへの攻撃手法として知られていた故障注入攻撃であるが，近年では，汎用ソフトウェアへの攻撃に適用可能であることが報告されている．故障注入攻撃を汎用ソフトウェアに適用した既存研究は，筆者の知る限り，4 件存在している．以上の既存研究で使用されている故障注入手法と故障モデルを表 2.5 にまとめる．故障モデルとは，故障による影響をモデル化したものである．実機に対して故障注入攻撃を行った既存研究での報告が多い，アセンブリレベルでの命令スキップや特定の 1 ビットの反転，あるデータの全ビットが 0，あるいは 1 になるといった現象が故障モデルとして利用されている．故障モデルを立てることにより，実機を用いた実験を行わずに，そのモデルに基づく攻撃手法の実現可能性を検証することや効果的な対策手法を考案することが可能となる．

文献 [15], [16], [17] は Java Virtual Machine (JVM) への攻撃のために故障注入攻撃を使用しているが，故障の注入手法や故障モデルは異なっている．文献 [15] では，攻撃対象機器を 50 ワットのスポットライトで 80〜100 度に熱することでデータのビット反転を引き起こしている．これにより JVM の型システムを誤らせることで，JVM バイトコード検証器によるコード検証を無効化し，任意コードを実行する．文献 [16] においても同様のアイデアで任意コードの実行を行うが，レーザ照射による命令スキップを用いている点が異なる．また，文献 [17] では，スタックメモリに格納されたリターンアドレスをレーザ照射により直接変化させることで，任意コードへ制御を移す．一方，JVM 以外への攻撃も報告されている．文献 [18] では，クロックグリッチを用いて，汎用ソフトウェアにおいて，関数呼び出し時のスタックポインタの更新をスキップすることで，BOF と同様の効果を発生させる．これにより，通常の BOF 攻撃と同様に任意コードの実行を行う．

ソフトウェアの設計時には，前後関係がなく突然データが異常な値になることや命令がスキップされることは考慮されていないため，ビット反転や命令スキップといった現象を引き起こすハードウェアへの攻撃に対しては脆弱である．そのため，表 2.5 に示した既存研究で報告されているように，JVM のバイトコード検証やセキュアコーディングといったソフトウェアレベルでの攻撃対策が無効化される可能性がある．また，表 2.5 に挙げた研究では，単一の故障のみで攻撃可能であるという特徴がある．単一故障の利点としては，一度の故障で攻撃可能であるため，攻撃の成功確率が高いということや，比較的単純な装置で実行可能であることが挙げられる．そのため，故障を注入する機器の精度はある程度低くても良いため，スポットライトで熱すること [15] やカメラのフラッシュ [40] で故障を誘発するような手法でも攻撃可能となる．

一方，近年では，暗号ソフトウェアへの攻撃において，複数回の故障を注入する多重故障注入を利用した攻撃が報告されている [19], [20], [41]．多重故障注入を利用した暗号ソフトウェアへの攻撃の報告により，プロセッサへ多重に故障を注入した際の影響が明らかになり，同攻撃の有効性が認識されてきている．これらの研究では，従来の単一故障注入攻撃への対策を施した暗号ソフトウェアに対する攻撃が報告されている．暗号ソフトウェアへの攻撃では，暗号処理中に故障を注入し，誤った暗号文を解析することで秘密鍵を不正に取得する．そのため，暗号文を出力する前に復号し，元の平文と等しいことを確認する等の検算による対策が一般的である．多重故障注入を利用した攻撃では，通常の攻撃と同様に一度目の故障で暗号処理を誤らせ，さらに二度目の故障で検算対策を無効化することにより誤った暗号文を出力させる．そのようなアイデアに基づいた，RSA への攻撃 [19], [41] や，AES への攻撃 [20] が報告されている．

このように，単一故障を利用した攻撃に対しては有効とされる対策であっても，多重故障を用いることで無効化される危険性がある．そのため，単一故障注入攻撃と比べて多重故障注入攻撃は，より強力な攻撃であると言える．そこで，汎用ソフトウェアに対する故障注入攻撃に関しても，多重故障注入を用いることで新たな攻撃の危険性が生じることが予測できる．本論文では，多重故障注入を用いた汎用ソフトウェアへの攻撃の実現可能性を検討する．また，多重故障注入のためには，故障タイミングを細かく設定可能で，実験の再現性も高い故障注入手法が必要となることから，表 2.4 のクロックグリッチを用いる．文献 [20] において，暗号プロセッサへの多重故障注入攻撃実験により，既にその有効性が実証されていることから，本実験では，特に，文献 [42] で提案された実装方法によるオンボード型のクロックグリッチを用いる．文献 [42] の手法では，FPGA 上にクロックグリッチ生成器を実装するため，クロックグリッチ生成用の専用の回路や複数の装置を要さないという利点がある．さらに，FPGA への制御信号により故障のパラメータを細かく設定できるため，一度実装した後は PC から制御可能である．そのため，試行回数が多くなることが予想される実験に向いている．基本的なアイデアは，図 2.3 に示すよう

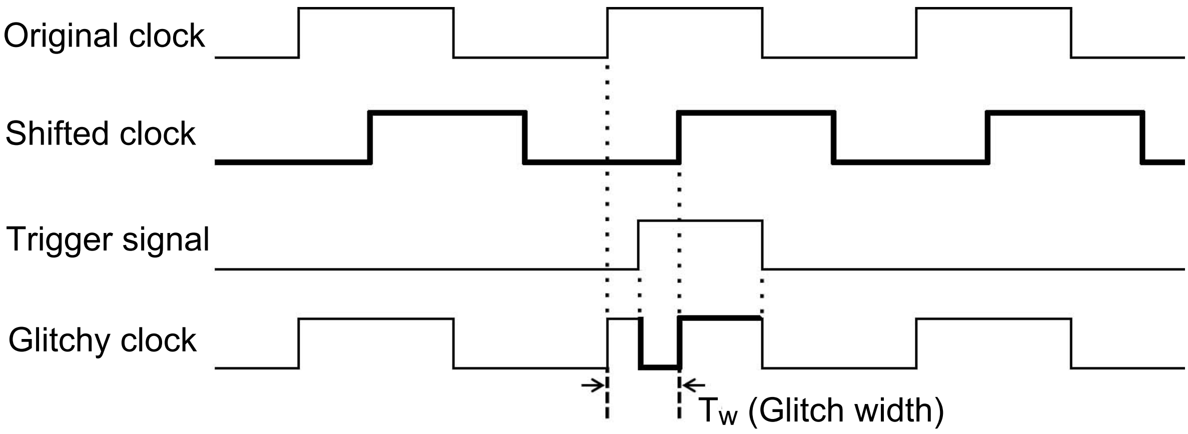


図 2.3　クロックグリッチの生成原理

に，位相の異なる 2 種類のクロック信号 (Original clock, Shifted clock) を生成し，任意のタイミング (Trigger signal) で切り替えることである．これにより，1 クロックの立ち上がりが二つ分割される．前半部分のクロック周期は非常に短くなっているため，フリップフロップのセットアップ時間違反が発生し，マイコンでは異常な動作が観測される．例えば，マイコンにおけるクリティカルパスはメモリアクセスであることが多いことから，命令のフェッチがスキップされ，命令が NOP (No operation) に変化したような動作をすることが報告されている [8], [20], [38]．そのため，クロックグリッチを用いた故障注入攻撃に関しては命令スキップの故障モデルが一般的に用いられているが，この他にも，図 2.3 のグリッチ幅 (Tw) を変化させることで，マイコン上で生じる故障の種類は変化する．このようにして，故障モデルと適合する故障を実験的に得ることが一般的である．

2.3.4　故障注入攻撃への対策

回路に入る故障への対策に関しては，フォールトトレランスの分野において，古くから研究が行われてきた [43], [44]．フォールトトレランスとは，システムの一部に故障が発生した場合であっても，正常時と同じようにサービスを提供し続けることが可能である，というシステムの耐障害性のことを指す．一般的には，宇宙線や環境温度などの外部的要因や，半導体接合の破壊や断線などの内部的要因が故障の発生原因として想定されている．そのため，複数の箇所が同時に故障する（多重故障）確立は低いと考えられている．こうした想定を置いて故障をモデル化することで，効果的に対策を適用し，コストを削減する．フォールトトレランスの基本原理は冗長性と分散である．冗長化手法としては，TMR (Triple Modular Redundancy, 三重冗長化) などの要素の冗長化や，演算処理を複数回実行するなどの時間の冗長化が挙げられる．また，そうして冗長化したシステムが同

時に故障することがないように，単一故障の影響下からシステムを分離する（分散する）必要がある．

フォールトトレランスの枠組みで提供される対策は，特定の故障原因だけに限定したものではなく一般性があるため，故障注入攻撃に関しても有効である場合も少なくない．しかし，フォールトトレランスでは，多くの場合，故障の発生が互いに独立であるという前提のもと，ある任意の瞬間に発生する故障は単一故障のみであると仮定している一方で，故障注入攻撃は，攻撃者が意図的に実行するものであり，任意のタイミングで多重に故障を発生させることが可能である．こうした理由から，フォールトトレランスにおいても多重故障に対応する必要があるが，そのために必要なコストは非常に高くなる．また，ローエンドなマイコンにおいては，その用途から考えて，フォールトトレランスの目的である，完全なサービスを維持し続ける，という対策水準は過剰であり，コストも見合わない．そこで，従来の対策と同様に，故障注入攻撃に特化した対策を検討する必要がある．

故障注入攻撃への対策はハードウェアによるものとソフトウェアによるものの二つに分けることができる．ハードウェアによる対策では，グリッチの検出器や DC フィルター，測温体を設けることや，そもそも外部からの電源やクロックの供給を禁止した設計にすることが挙げられる．しかし，こうした対策もフォールトトレランスと同様に，コスト的な問題から実装は難しい．また，一般的に，汎用マイコンを使用した場合にはアーキテクチャの変更が困難であるという点で汎用性に欠ける．さらに，光電効果を利用した光の照射のように，LSI の物理的な特性に基づく故障注入手法や未知の故障注入手法に対しては，ハードウェアによる対策だけで防ぐことは難しい．

従って，特定の故障モデルに基づく攻撃への対策を講じるソフトウェアでの対策が，故障注入攻撃への対策手法としては有効であると考える．暗号ソフトウェアによる攻撃対策では，誤った暗号文を出力しないことが求められる．そのため，検算処理を用いて故障が注入されたことを検出し，暗号処理を停止する対策や [37], [45]，故障注入により命令がスキップされても暗号文を出力しないようなルーチンへ変更する対策 [20] が挙げられる．また，短い時間間隔で連続して故障を注入することは難しいという想定の下，命令を冗長化する対策 [46], [47] や，符号化により誤りを検出する対策 [37], [47] も存在する．

2.4　結び

本章では，組込みソフトウェアへの攻撃に関する基礎的考察を行った．まず，ソフトウェアへの論理攻撃の中でも，報告件数が多い攻撃について述べ，特に，組込みソフトウェアへの攻撃としても関心が高い BOF 攻撃に着目した．BOF 攻撃の危険性について述べ，攻撃原理と対策手法について述べた．次に，ハードウェアへの物理攻撃の分類を行い，特に，ソフトウェアへの攻撃として有効であると考えられる故障注入攻撃に着目し

た．故障注入攻撃を故障の種類や故障の注入手法に応じて分類し，また，汎用ソフトウェアへの攻撃のために用いられている手法をまとめた．さらに，多重故障注入の危険性を指摘し，本論文で行う実験にはクロックグリッチが適切であることを述べ，同故障注入手法の実装方法を述べた．最後に，故障注入攻撃への対策について，フォールトトレランスと従来の故障注入攻撃への対策手法に分けて述べた．

第 3 章

故障注入を併用したバッファオーバーフロー攻撃

3.1 はじめに

本章では，故障注入攻撃を併用した BOF 攻撃を提案する．まず，関連研究との比較を行うと同時に，提案攻撃の基本的なアイデアを述べる．次に，AVR マイコンと ARM マイコンを例として，それぞれのアーキテクチャ，および提案攻撃の適用方法について述べる．

3.2 関連研究

本論文では，入力サイズの制限を行うことが可能な関数を利用することを BOF 攻撃対策として想定し，故障注入により生じるアセンブリレベルでの命令スキップを利用して，同対策を無効化する攻撃手法を提案する．故障注入を用いたソフトウェアへの攻撃を報告している既存研究の中でも，特に文献 [18] は，BOF に着目している点と命令スキップを利用している点で，本研究と類似する．文献 [18] の基本的なアイデアは，スタックポインタの値を不正な値に変更することで，正常な場合とは異なるアドレスからリターンアドレスを取得させることである．そのために，まず，関数呼び出しの発生時に実行されるスタックポインタの更新命令をスキップする．これにより，スタックポインタは変更されずに呼び出された関数が実行される．そのため，呼び出した関数の終了時には，誤ったスタックポインタを用いてリターンアドレスが取得される．こうして，リターンアドレスは不正なアドレスから取得される．このアドレスにあらかじめ任意のアドレスを示すデータを配置しておくことで，任意の関数を実行可能となる．このように，メモリ上に配置したデータをプロセッサに解釈させ，任意のアドレスへ制御を移す点で，BOF と同じような効果と言える．この攻撃の利点として，注入する故障は一度で良いことと，関数呼び出しが存在する場合には実行可能であるため，あらゆるプログラムが攻撃対象となることが挙げられる．

しかし，この攻撃には二つ，大きな問題点が存在する．一つは，攻撃者はスタックポインタの変化量を操作できないため，攻撃が実行可能である確立が低いことである．スタックポインタの更新における変化量は呼び出した関数の構造に依存するため，プログラムが作成された時点で決定され，攻撃者が操作することはできない．もう一つは，任意の関数を呼び出す前にプログラムが異常停止する可能性があることである．スタックポインタを不正な値に変更した後，そのスタックポインタ相対でリターンアドレスが取得される前には，必ず呼び出した関数が実行される．そのため，この関数の中で，スタックポインタ相対でのメモリ参照により異常な値のロードやストアが行われた場合，プログラムは異常停止する．一方で，本論文で提案する手法では，BOF 攻撃対策を無効化することで BOF を発生させ，これを利用してリターンアドレスを書き換えるという通常の BOF 攻撃を実行する．そのため，BOF 攻撃手法に関する既存研究の成果を応用することもでき，攻撃の影響はより大きいと言える．

3.3　AVR マイコンへの適用

本節では，8 ビット AVR マイコンに対して提案攻撃を適用する方法について述べる．ここでは，一般的な 8 ビット AVR マイコンの例として，サイドチャネル攻撃標準評価用ボード SASEBO (Side-channel Attack Standard Evaluation BOard)-W[48] で用いられるスマートカード上に搭載されている ATmega163 に着目する．まず，ATmega163 のアーキテクチャについて述べる．その後，攻撃対象として想定する関数のアセンブリ命令を解析することを通して，BOF 攻撃対策を無効化する方法について述べる．

3.3.1　ATmega163 アーキテクチャ

ATmega163 は，ハーバードアーキテクチャを採用した 8 ビット RISC アーキテクチャである [49]．そのため，プログラムメモリとデータメモリが物理的に分離しており，データメモリに格納した攻撃コードを実行することは不可能となっている．しかし，リターンアドレスを書き換えて任意の関数を呼び出すことは可能であるため，AVR マイコンに対する BOF 攻撃の脅威は小さくないと言える．また，1 サイクルに Fetch と Execute を行う単純な実装となっている 2 段のパイプライン構造を有している．

レジスタセットは，32 個の 8 ビット汎用レジスタ (R0 - R31) の他，プログラムカウンタ (Program counter: PC)，スタックポインタ (Stack pointer: SP)，ステータスレジスタからなる．汎用レジスタの内，R18 - R25 は関数呼び出しの際の引数や戻り値として使用される．引数や戻り値は R25 から R18 へと順に格納され，足りない分はスタックに格納される．サブルーチンの呼び出しは，`CALL` 命令を用いて実行される．この時，リ

ターンアドレスをスタックに保存する処理やスタックポインタの値の更新は，同命令の中で自動的に行われる．また，サブルーチンからの復帰には `RET` 命令が使用される．この命令では，スタックから取得したリターンアドレスを PC へとセットすると同時にスタックポインタの値の更新も行う．

また，レジスタ R26 - R31 には特殊な機能が与えられている．R27:R26, R29:R28, R31:R30 の 3 組のレジスタペアはそれぞれ X, Y, Z レジスタと呼ばれ，「上位ビット:下位ビット」のように，16 ビットとして扱われる．こうした特殊レジスタは，主に，メモリへの相対アクセスの際に用いられる．相対アクセスには，X レジスタ等を用いた単純な相対アクセスの他に，事前減少や事後増加，固定オフセット付きの 4 種類のアクセス法が定義されている．なお，Y レジスタには，関数呼び出しが発生した時点のスタックポインタの値を格納するフレームポインタとしての役割も存在する．

3.3.2　ATmega163 への適用

2.2.3 節で述べたように，想定する対策は入力データのサイズ制限が可能な関数を使用することである．サイズ制限部分の命令をスキップすることで，この対策を無効化可能であると予想できる．ここでは，具体例として，`strncpy()` を用いて文字列を処理するプログラムを想定する．この関数は，`strncpy(char *dest, const char *src, size_t size)` として定義されており，`src` から `dest` へ，最大で `size` 分だけ文字列をコピーする．なお，`src` が `size` に満たない場合には，不足分だけ 0 埋めを行う．

まず，故障注入によりスキップする命令を決定する．avr-libc (1.6.7) のライブラリセットに基づく `strncpy()` のアセンブリコードを図 3.1 に示す．処理の流れは大きく二つに分けることができる．一つは，文字列のコピーを `size` 回だけ繰り返す処理（4-12 行）である．もう一つは，`size` 回のコピーが終了する前に NULL 文字 (0x00) が現れた場合に，`size` に満たない分を 0 埋めする処理（13-18 行）である．BOF 攻撃対策と関連する部分は，`size` 回のコピーが行われたことを確認して終了する処理である．また，BOF 攻撃を行うためには，ユーザ入力をメモリに格納する必要があることから，このコピー処理のみが関心部分となる．

以上を踏まえて前半部分の命令をより詳細に分析する．図 3.1 の 5, 6 行目の `SUBI`, `SBCI` 命令では，R21:R20 にセットされた計 16 バイトのループカウンタの減算を実行する．このループカウンタが `strncpy()` の引数 `size` に相当する．`BRCS` 命令では，(`size` + 1) 回の減算時に `RET` へ分岐する．`LD` 命令では，Z レジスタ（R30:R31）の値を使用した相対アドレス指定により，コピー元のアドレスから 1 文字 (1 バイト) だけ，R0 レジスタへコピーする．この時，同時に Z レジスタの値をインクリメントしてコピー元の位置をずらし，次の文字をコピーする準備を行う．`ST` 命令では，先ほど R0 レジスタにコピーし

```
1  strncpy:
2      MOVW R30, R22    ;  •Z レジスタに引数 1 をセット
3      MOVW R26, R24    ;  •X レジスタに引数 2 をセット
4  LOOP:    ; メモリのデータコピーを行うループ
5      SUBI R20, 0x01   ;  •ループカウンタの
6      SBCI R21, 0x00   ;    減算,および
7      BRCS RET         ;    チェック(0か?)
8      LD   R0, Z+      ;  •データをロード
9      ST   X+, R0      ;  •データをストア
10     AND  R0, R0      ;  •NULL 文字（0x00）の
11     BRNE LOOP        ;    チェック
12     RJMP ZERO
13 LOOP_Z:    ; 以下,余ったバッファの 0埋め処理
14     ST   X+, R1
15 ZERO:
16     SUBI R20, 0x01
17     SBCI R21, 0x00
18     BRCC LOOP_Z
19 RET:
20     RET
```

図 3.1　strncpy() のアセンブリコード（AVR）

た文字を，X レジスタ（R26:R27）の値を使用した相対アドレス指定によりコピー先アドレスへコピーする．こちらも，次の文字列をコピーする準備として，X レジスタの値をインクリメントする．AND 命令では，R0 同士の論理積を取る．もしも R0 に入った文字が NULL 文字 (0x00) ならば，論理積の計算結果は 0 となりゼロフラグ (ZF) が 1 になる．これにより，次の BRNE 命令では分岐が起こらず，後半部分の size に満たないバイトの 0 埋め処理に移行する．R0 に入った文字が NULL 文字以外ならば，この BRNE で分岐が発生して 5 行目に戻り，以上の処理が繰り返し実行される．

以上の分析から，BOF 攻撃対策を無効化する方法を 2 通り挙げることができる．一つは，5 行目のループカウンタの減算命令 (SUBI) をスキップする方法である．減算命令をスキップするごとに 1 ループ多く回ることになるため，1 文字分だけ入力数が増える．もう一つは，7 行目の size 回の文字コピー終了後の RET への分岐命令 (BRCS) である．この命令をスキップしても，同様の効果が得られる．8 ビットレジスタにおいては，0 から 1 を減じると，値は 0xFF となることから，分岐命令をスキップする場合には，ループカウンタが 0 になってから一度だけ故障を入れるだけで良い．本論文では，多重故障注入攻撃に基づく攻撃の実現可能性を実証することが目的に含まれているため，分岐命令のスキップに関しては考慮しない．そこで，AVR マイコンでは，減算命令を繰り返しスキップすることにより，BOF 攻撃対策の無効化を試みる．このために，各ループ処理の中で，

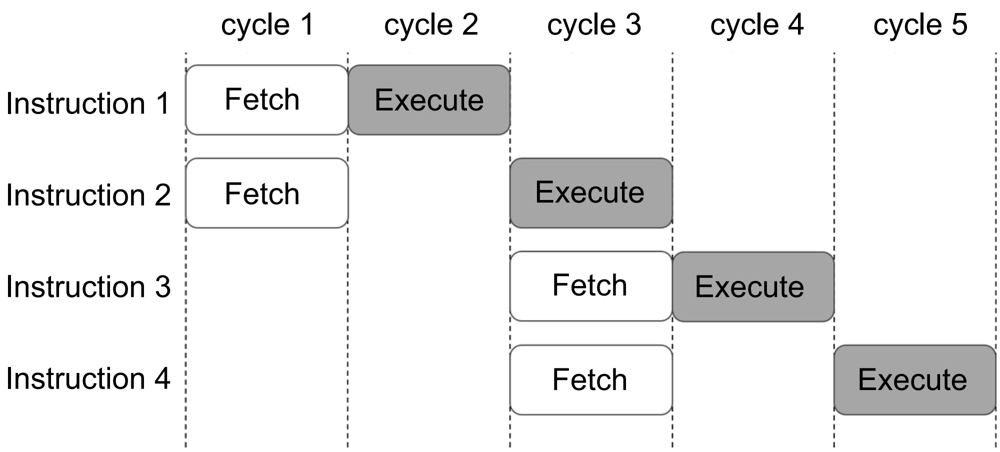


図 3.2　Cortex-M0+ のパイプライン構造

同じタイミングで故障を入れる必要がある．図 3.1 のループ処理 (4-12 行目) が終了せずに繰り返される時，7 行目の `BRCS` は分岐せず (`size` 回に満たない)，11 行目の `BRNE` は分岐する (NULL 文字ではない)．この条件において，1 ループあたり 10 クロックサイクルを要する．従って，`strncpy()` が呼ばれ，減算命令が実行され始めるタイミングで最初の故障を入れ，その後は 10 クロックサイクルごとに繰り返し故障を入れることで，減算命令のみを繰り返しスキップすることができる．

3.4　ARM マイコンへの適用

本節では，ARM マイコンの中でも，特に，ローエンド向けに設計されている Cortex-M0+ ベースのマイコンに着目をする．まず，Cortex-M0+ アーキテクチャについて述べる．その後，アセンブリ命令の解析を行うことで，BOF 攻撃対策を無効化する方法について述べる．

3.4.1　Cortex-M0+ アーキテクチャ

ARM Cortex-M0+ はフォン・ノイマンアーキテクチャを採用した 32 ビット RISC アーキテクチャであり，2 段パイプラインを採用して分岐予測の際の無駄を減らすなど，エネルギー効率を最重視したモデルとなっている [50]．命令セットは Cortex-M0, M1 と同じく，Thumb 命令および一部の Thumb-2 命令を含む 56 個の命令が使用可能である．命令は，図 3.2 に示すように，Fetch と Execute という 2 段パイプライン上で実行される．実行される命令の多くは 16 ビットで構成される一方，バス幅は 32 ビットであるため，Fetch ステージでは，二つの命令を同時に取得し，メモリアクセスによって生じる消

費電力を抑えている．

レジスタセットは，13 個の汎用レジスタ (R0 - R12) の他，サブルーチンや例外からの復帰情報を格納するリンクレジスタ (Link register: LR (R14))，SP (R13)，PC (R15)，プログラムステータスレジスタとなっている．関数呼び出しで引数を渡す場合は，第 1 引数から第 4 引数までが，それぞれ R0 から R3 までの各レジスタに格納される．また，戻り値は R0, R1 に格納される．これらのレジスタで不足が生じた場合には，スタックを用いて値の受け渡しを行う．関数呼び出しは `BL` (Branch with Link) 命令が使用される．`BL` は分岐と同時にリターンアドレスを LR へと格納する．LR は呼び出した関数の中で，`PUSH` 命令を用いてスタックへの退避が必要となる．また，そうして退避したリターンアドレスは関数呼び出しの終了時に `POP` 命令を用いて PC へと書き戻される．

3.4.2 Cortex-M0+ マイコンへの適用

ARM マイコンに関しても，AVR マイコンと同様にスキップするべき命令を決定する．まず，Newlib (1.19) に基づく `strncpy()` のアセンブリ命令を図 3.3 に示す．ARM のアセンブリコードも，AVR のものと同じ構造を取っており，処理の流れは大きく二つに分けることができる．前半の文字列コピー処理（4-13 行）と後半の 0 埋め処理（14-22 行）である．従って，関心部分は前半の文字列コピー処理のみである．なお，AVR マイコンとの主な違いは次の通りである．

1. ループカウンタは 32 ビットの 1 レジスタ (R2) により構成される
2. 任意レジスタを用いた相対アドレス指定が可能 (7, 9, 20 行目)
3. データのロード (ストア) 命令とロード (ストア) アドレスの更新を分離 (7, 10 行目，および 9, 11 行目)

こうした違いは攻撃手法に影響を与えないため，BOF 攻撃対策を無効化する方法も同様である．入力サイズ制限の要となる処理は，(1) 5, 6 行目のループカウンタの値をチェックし，カウンタの値が 0 であれば 0 埋め処理に移行する部分と，(2) 8 行目のループカウンタの減算を行う部分である．従って，ARM マイコンへの攻撃実験では，これらの処理を攻撃対象とする．

3.5 結び

本章では，故障注入を併用したバッファオーバーフロー攻撃について述べた．まず，故障注入攻撃とバッファオーバーフローに着目している点で本研究と類似する文献 [18] の研究と比較を行うと同時に，提案攻撃の基本的なアイデアを述べた．次に，8 ビット AVR

```
strncpy:
    PUSH {R4, LR}
    ADDS R3, R0, #0
1:    // メモリのデータコピーを行うループ
    CMP  R2, #0          // •ループカウンタの
    BEQ  2f              //   チェック(0か?)
    LDRB R4, [R1, #0]    // •データをロード
    SUBS R2, #1          // •ループカウンタ--
    STRB R4, [R3, #0]    // •データをストア
    ADDS R1, #1          // •ロードアドレス ++
    ADDS R3, #1          // •ストアアドレス ++
    CMP  R4, #0          // •NULL 文字(0x00)の
    BNE  1b              //   チェック
2:    // 以下,余ったバッファの 0埋め処理
    ADDS R2, R3, R2
3:
    CMP  R3, R2
    BEQ  4f
    MOVS R1, #0
    STRB R1, [R3, #0]
    ADDS R3, #1
    B    3b
4:
    POP  {R4, PC}
```

図 3.3　strncpy() のアセンブリコード（ARM）

マイコンのアーキテクチャについて述べ，strncpy() を対象として，そのアセンブリ命令の解析を行い，提案攻撃を適用する方法について述べた．その後，ARM マイコンの中でもエネルギー効率を最重視した，ローエンド向けの Cortex-M0+ のアーキテクチャについて述べ，Cortex-M0+ ベースのマイコンに対して提案攻撃を適用する方法について述べた．こうした検討を通して，一般的に，攻撃対象のプログラムが本提案攻撃に対する脆弱性を有する条件は，(1) ループ構造を有すること，(2) ユーザ入力がメモリからロードされること，(3) そのデータがメモリへとストアされること，という三つの前提を満たすことだと言える．こうした脆弱性を持つ場合，(1) ループカウンタの更新，および (2) 入力サイズ制限に基づく終了判定，という二種類の処理が命令スキップの対象となる．

第 4 章

評価実験と対策の検討

4.1 はじめに

本章では，AVR マイコンと ARM マイコンの 2 種類のマイコンを使用した実験を通して，第 3 章で述べた提案手法の有効性を実証する． まず，AVR マイコンと ARM マイコンのそれぞれに対して，攻撃実験を行った結果を述べる．その後，各マイコンにおける対策手法について述べる．

4.2 AVR マイコンへの攻撃

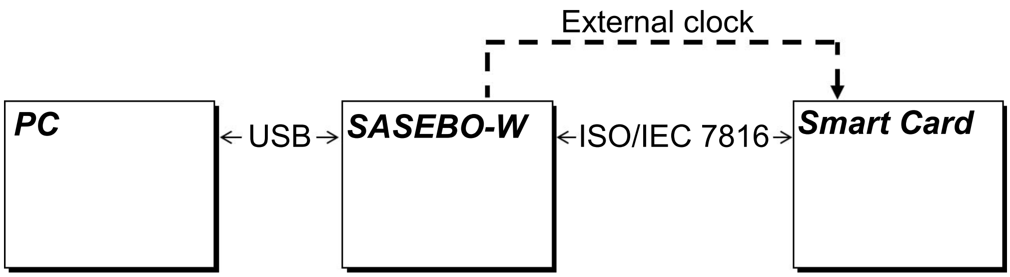
本節では，AVR マイコンに対して攻撃実験を行った結果を述べる．まず，実験環境について述べ，故障を注入せずにプログラムを正常に動作させた時のメモリ情報に基づき，攻撃コードの作成を行う．次に，故障注入を併用して攻撃コードの送信を行い，任意関数の呼出しを行う．

4.2.1 実験環境

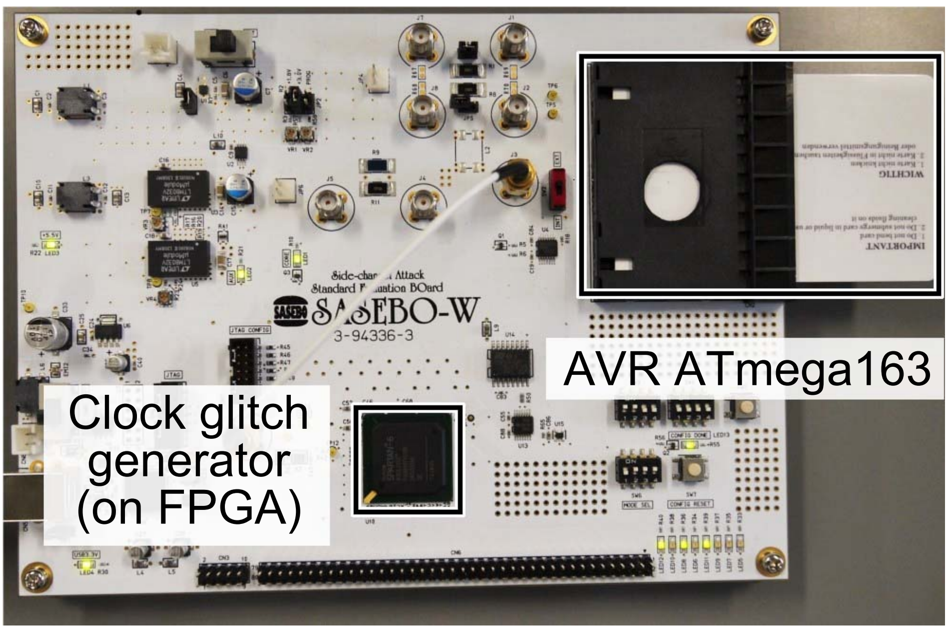
実験条件および実験環境をそれぞれ表 4.1 と図 4.1 に示す．実験には, 図 4.1 に示すサイドチャネル攻撃標準評価用ボード SASEBO (Side Channel Attack Standard Evaluation BOard) -W[48] を使用した．このボードで使用されるスマートカード上に 8 ビットマイコン AVR ATmega163 が搭載されており，同マイコン上でプログラムが動作する．SASEBO-W 上の FPGA には文献 [42] の手法によるオンボード型のクロックグリッチ生成器が実装されており，PC からの制御信号に応じて，スマートカードへとグリッチ入りクロックを供給する．また，本 FPGA は通信の制御も行っており，PC からの制御信号を受け取りクロック信号の生成を行う他，PC とスマートカード間のデータの受け渡しも行う．なお，攻撃条件として，プログラムは既知であるとした．

表 4.1　実験条件 (AVR)

マイクロコントローラ	AVR ATmega163
動作周波数	3.6 MHz (許容値: 0〜8MHz)
コンパイラ	avr-gcc (4.3.3) (最適化オプション -O0)
ライブラリ	avr-libc (1.6.7)
FPGA	Xilinx XC6SLX150
グリッチ幅 (Tw)	28.2 ns



(a) ブロック図



(b) 全景

図 4.1　実験環境 (AVR)

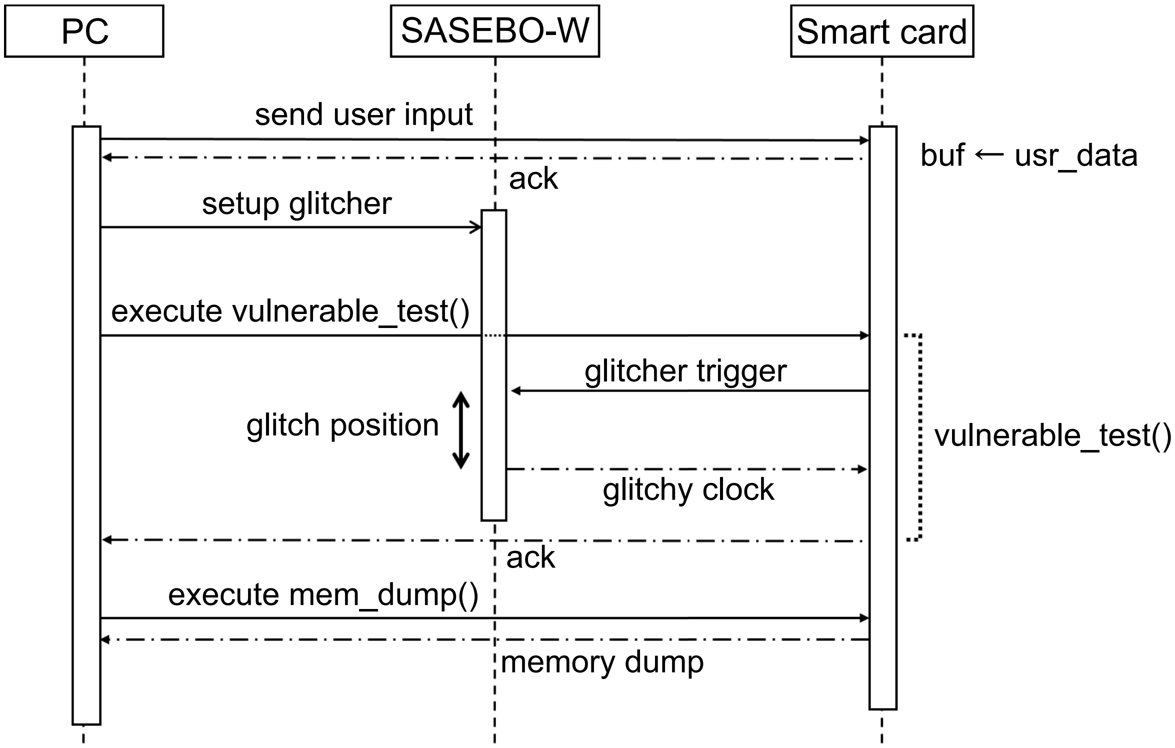


図 4.2　プログラムフロー (AVR)

実験で使用したプログラムのフローを図 4.2 に示す．図中の実線の矢印は操作の呼出しであり，点線の矢印がその応答である．応答が無いものは非同期通信である．まず，PC から 32 バイトのユーザ入力を受け取り，32 バイトのグローバル配列変数 `buf` へと格納する．その後，PC からの制御信号により，グリッチ位置やグリッチ幅 (Tw) といったグリッチパラメータを設定する．続いて，図 4.3 に示す `vulnerable_test()` を呼び出す．スマートカードでは，`vulnerable_test()` の呼び出しと同時に，トリガ信号 (glitcher trigger) を SASEBO-W へと送信する．このトリガ信号を受け取ったタイミングを position=0 [cycle] とし，先ほど設定したグリッチ位置に到達したサイクル数にグリッチを入れる．図 4.3 の `vulnerable_test()` では，20 バイトの配列変数 `msg` が宣言されており，`strncpy()` を用いて最大で 19 バイト分だけ `buf` から `msg` へとコピーする．その後，呼び出される `stack_dump()` は，攻撃の影響を解析するためにメモリ内容を保存する関数である．`vulnerable_test()` の実行後，データメモリの内容を取得する `mem_dump()` を呼び出して，メモリダンプを取得する．

AVR マイコンを用いた実験では，故障注入を併用して BOF 攻撃を行い，スタック上のリターンアドレスを書き換えることで図 4.4 に示す関数を不正に呼び出す．この関数は，`stack_dump()` でメモリ内容の保存先の一部となっているアドレス 0x120 へ “Hello

```
void vulnerable_test(void) {
    uint8_t msg[20] = {0};

    strncpy(msg, buf, sizeof(msg) - 1);

    stack_dump(&msg[0], 0x00e0, 32, 64);
}
```

図 4.3　vulnerable_test() の C コード (AVR)

```
void hello_world(void){
    memcpy((uint8_t *)0x120, "hello world!", 12);
}
```

図 4.4　不正に呼び出す関数

```
00e0  22 a0 05 10 06 98 04 06 08 04 01 00 1c 23 16 19  "            #
00f0  01 b6 01 04 21 04 e0 00 20 00 40 00 20 04 03 73      !     @    s
0100  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0110  41 41 41 00 38 04 03 82 3c 04 05 f6 00 bf 00 53  AAA 8   <      S
0120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

図 4.5　故障注入なし，20 バイトの文字列送信時のメモリダンプ

world!” という文字列を書き込む．この文字列の有無により，攻撃が成功したかどうかを判断する．攻撃の手順は，攻撃コードの作成，故障注入回数の決定，攻撃の実行，という流れになる．

4.2.2　攻撃コードの作成

実験で使用する攻撃コード作成のため，書き換えるべきスタックの内容を調べる．図 4.3 の攻撃対象のコードは 19 バイトの入力を受け取る仕様となっている．そこで，まずは故障注入を行わずに，“A” が 20 個続く 20 バイトの文字列を送り，これを読み込ませた時のスタックの状態を調べる．この時のメモリダンプを図 4.5 に示す．左から，メモリアドレス，16 進数表記でのメモリデータ，文字列表記でのメモリデータである．文字列表記から，アドレス 0x0100 から 0x0112 に，入力データの 19 バイト分がコピーされていることが分かる．このように，strncpy() では引数で設定した入力サイズまでしか読み込まないため，適切に引数の入力サイズを設定している限りは BOF が発生しない．また，

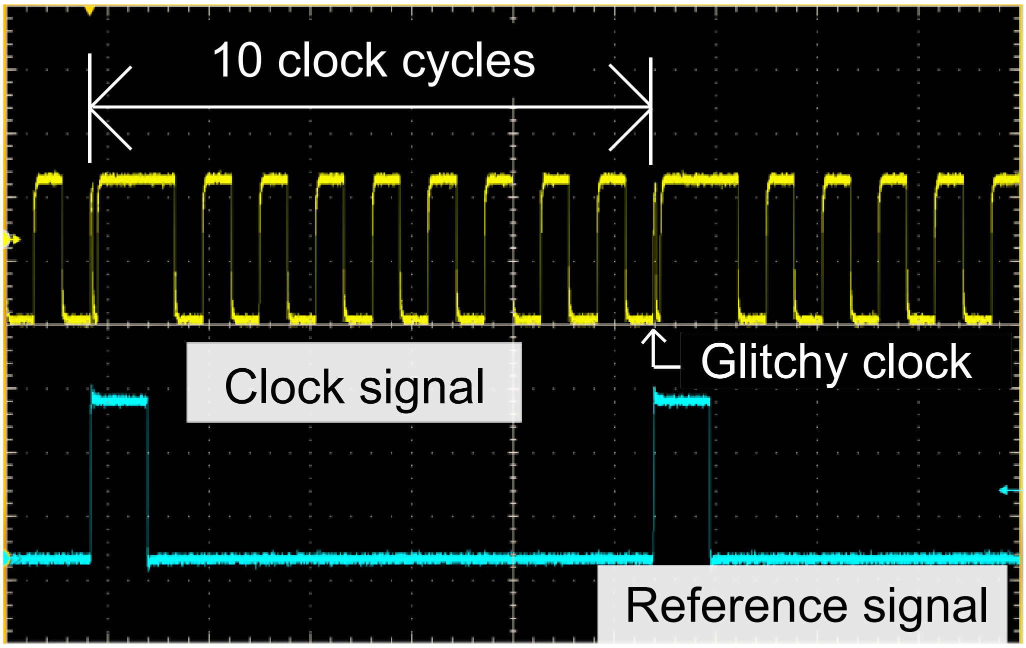


図 4.6　BOF 攻撃対策を無効化するためのクロックグリッチ

関数呼び出し時にスタックに格納される順番を考慮すると，この次の 0x0114 からの 2 バイト (0x3804) がスタックポインタ，0x0116 からの 2 バイト (0x0382) がリターンアドレスであることが分かる．従って，このリターンアドレスを，呼び出したい関数が配置されたアドレスで上書きすることが攻撃の目的となる．

次に，攻撃コードを作成する．先頭 20 バイト分は現在の値のまま，スタックポインタの値は実験から得られた 0x3804 という値に，リターンアドレスは図 4.4 のコードが配置された 0x0412 というアドレスに書き換える．図 4.4 のコードが配置されたアドレスは，プログラムのオブジェクトダンプにより得られた値を利用している．以上より，攻撃コードは，“A...A(0x38)(0x04)(0x04)(0x12)“(先頭は”A” が 20 個続く) という 24 バイトの文字列とする．また，現在，19 バイト分の文字列が 0x0112 まで読み込まれており，リターンアドレスが配置された 0x0117 まで読み込ませるためには，さらに 5 文字を読み込ませる必要がある．このために，クロックグリッチを用いて，5 回の減算命令を連続でスキップする．

4.2.3　攻撃実験

本実験で使用したクロック信号を図 4.6 に示す．上部の信号がクロック信号，下部の信号がリファレンス信号である．グリッチは，ユーザの設定したグリッチパラメータに応じ

```
00e0  22 a0 05 10 06 99 04 06 28 84 01 00 9c 03 56 19  "       (     V
00f0  01 b6 01 04 21 04 e0 00 20 00 40 00 20 04 03 73       !     @    s
0100  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0110  41 41 41 41 38 04 04 12 3c 04 05 96 00 bf 00 53  AAAA8   <      S
0120  68 65 6c 6c 6f 20 77 6f 72 6c 64 21 00 00 00 00  hello world!
```

図 4.7　故障注入あり，24 バイトの攻撃コード送信時のメモリダンプ

て発生するリファレンス信号の立ち上がりで挿入される．ここでは，実験的に求めた最初の減算命令の位置から，10 クロックサイクル毎に 5 回のグリッチを生成している．

クロックグリッチを使用しつつ，作成した攻撃コードを送信した結果を図 4.7 に示す．故障注入を行わなかった場合の図 4.5 とは異なり，0x0100 から 0x0117 まで読み込まれていることが確認できる．このように，`strncpy()` では 19 バイトまでしか読み込まないように制限しているにも関わらず，送信した 24 バイト全て読み込まれ，リターンアドレスが 0x0412 に書き換えられている．これにより，図 4.4 のコードが呼び出され，0x0120 から 0x012B の部分に”Hello world!” という文字が書き込まれていることが確認できる．

4.3　ARM マイコンへの攻撃

本節では，ARM マイコンに対して攻撃実験を行った結果を述べる．Cortex-M0+ ベースのマイコンに対する故障注入攻撃を行った既存研究は，筆者の知る限り，存在しないことや，8 ビット AVR マイコンと比べてアーキテクチャが複雑で，かつ動作周波数も高いことから，攻撃はより難化すると予想した．そこで，まず，故障感度解析実験を行い，3.4.1 節で述べた，パイプライン構造やフェッチの方法を考慮して，連続する 2 命令に対して命令スキップを引き起こした場合の影響を解析する．次に，故障感度解析実験で得られた結果を元に，攻撃のために必要な命令スキップのパターンを検討し，提案攻撃を実行する．

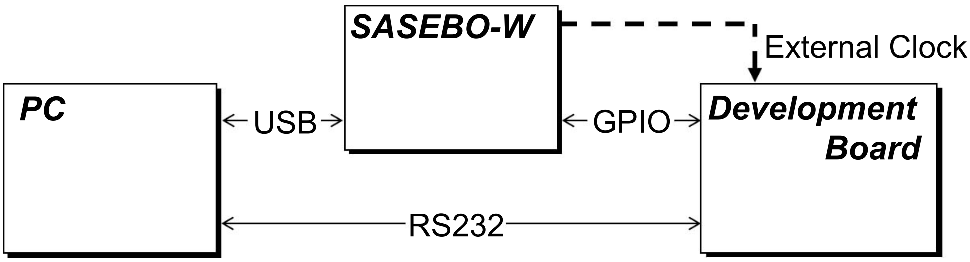
4.3.1　故障感度解析実験

実験条件および実験環境をそれぞれ表 4.2 と図 4.8 に示す．AVR マイコンへの攻撃実験とは異なり, SASEBO-W は FPGA の部分のみを使用しており, グリッチ入りのクロック信号を ARM マイコンへと供給する働きしか持たない．実験には, STMicroelectronics 社製の評価・開発用ボード STM32L053 Discovery[51] を使用した．本ボードには，Cortex-M0+ ベースのマイコン STM32L053C8 が搭載されている．

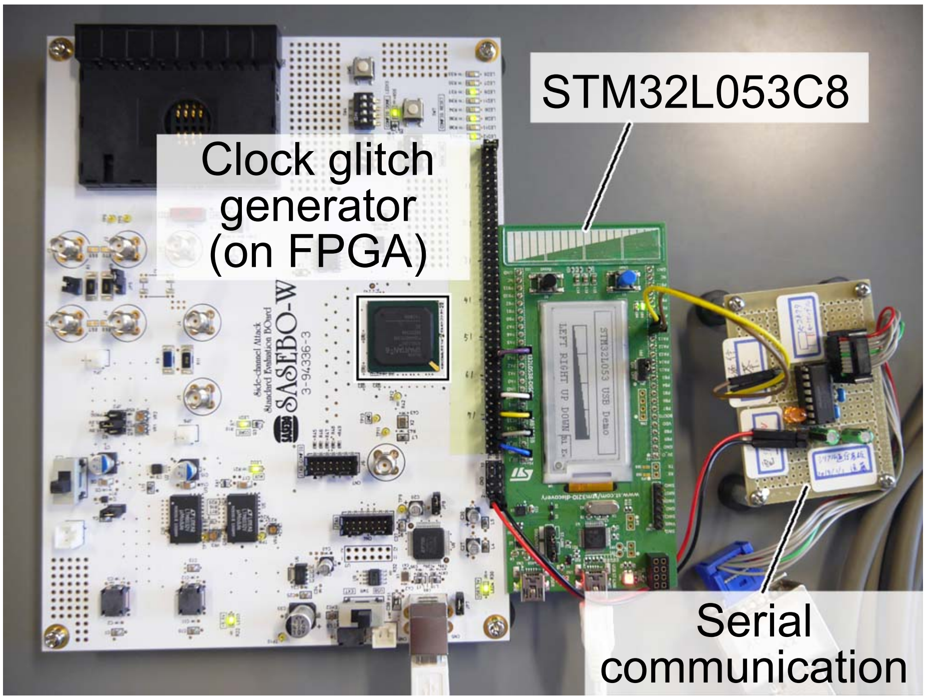
ARM マイコンへの攻撃実験に関しては，AVR マイコンと比べてアーキテクチャが複

表 4.2　実験条件 (ARM)

マイクロコントローラ	STM32L053C8
動作周波数	24MHz (許容値: 32KHz〜32MHz)
コンパイラ	arm-atollic-eabi-gcc (4.7.3) (最適化オプション -Os)
ライブラリ	Newlib (1.19)
FPGA	Xilinx XC6SLX150
グリッチ幅 (Tw)	11.7〜28.5 ns (0.2ns 間隔)



(a) ブロック図



(b) 全景

図 4.8　実験環境 (ARM)

雑であるため，まず，命令スキップの影響を解析し，その後，攻撃を行う．実験では，連続する 2 命令に対する故障注入により生じる影響の解析を通して，2 段パイプラインが命令スキップに及ぼす影響を解析する．対象命令は，算術演算命令 (ADDS, MULS)，メモリ操作命令 (STRB, LDRB)，分岐命令 (B) の 5 命令とする．単一命令 5 通りに，5 命令の重複順列 25 通りを加えた，計 30 通りのパターン（以下，命令グループと呼ぶ）について，故障の影響を解析する．ここで，ADDS, MULS は 1 クロックサイクル，STRB, LDRB, B は 2 クロックサイクルで実行されることに注意されたい．これにより Execute ステージの長さが変わり，実際には，図 3.2 のように単純な処理フローとならない．そのため，命令の組合せによって観測できる故障のパターンが異なることが予想できる．なお，各命令グループが実行される前後には NOP (No operation) 命令を挿入し，解析対象以外の命令に影響が出ないようにする．命令グループの二つの命令の並びを {inst1, inst2} と表し，inst1 に続いて inst2 が実行されるとする．故障の影響を，次の 5 種類に分類する．

$[0]$　故障は発生しない
$[1]$　inst1 のみスキップされる
$[2]$　inst2 のみスキップされる
$[1, 2]$　inst1, inst2 の両方がスキップされる
$[N/A]$　それ以外の解析不能な誤り

例として，命令グループ {ADDS, STRB} について故障感度解析を行った結果を図 4.9 に示す．横軸はグリッチ位置で，マイコンからの同期信号（トリガ信号）を FPGA が受信したタイミングを position=0 として数えたクロックサイクル数となる．縦軸はグリッチ幅 (Tw) である．各故障タイプと色の関係はそれぞれ，緑: $[0]$，赤: $[N/A]$，青: $[1]$，黄: $[2]$，シアン:$[1, 2]$ となる．連続する 2 命令のフェッチのされ方は，図 3.2 の Instruction1, Instruction2 のように同時にフェッチされるパターンと Instruction2, Instruction3 のように個別にフェッチされるパターンの 2 通りがある．{ADDS, STRB} の場合，図 4.10 のように表すことができる．四角で囲まれた F と E はそれぞれ，Fetch, Execute ステージを示す．なお，命令グループの前に位置する命令のサイクル数によっては Execute ステージの長さが変わるため，図 4.10 以外のパターンも存在する．今回の実験では，命令グループの前に位置する命令は NOP であるため，Execute ステージは必ず 1 クロックサイクルで終了することから，図 4.10 の 2 種類のみを考慮する．図 4.9 の結果は，position=33 からフェッチが始まったと仮定すると，図 4.10 (b) の 2 命令が個別にフェッチされるパターンと適合する．故障が注入されたタイミングは，それぞれ次のように推測できる．

I: ADDS (Fetch) (Tw $<$ 26[ns])
II: ADDS (待機) (Tw $<$ 27[ns])
III: ADDS (Execute), STRB (Fetch) (Tw $\approx$ 16[ns])

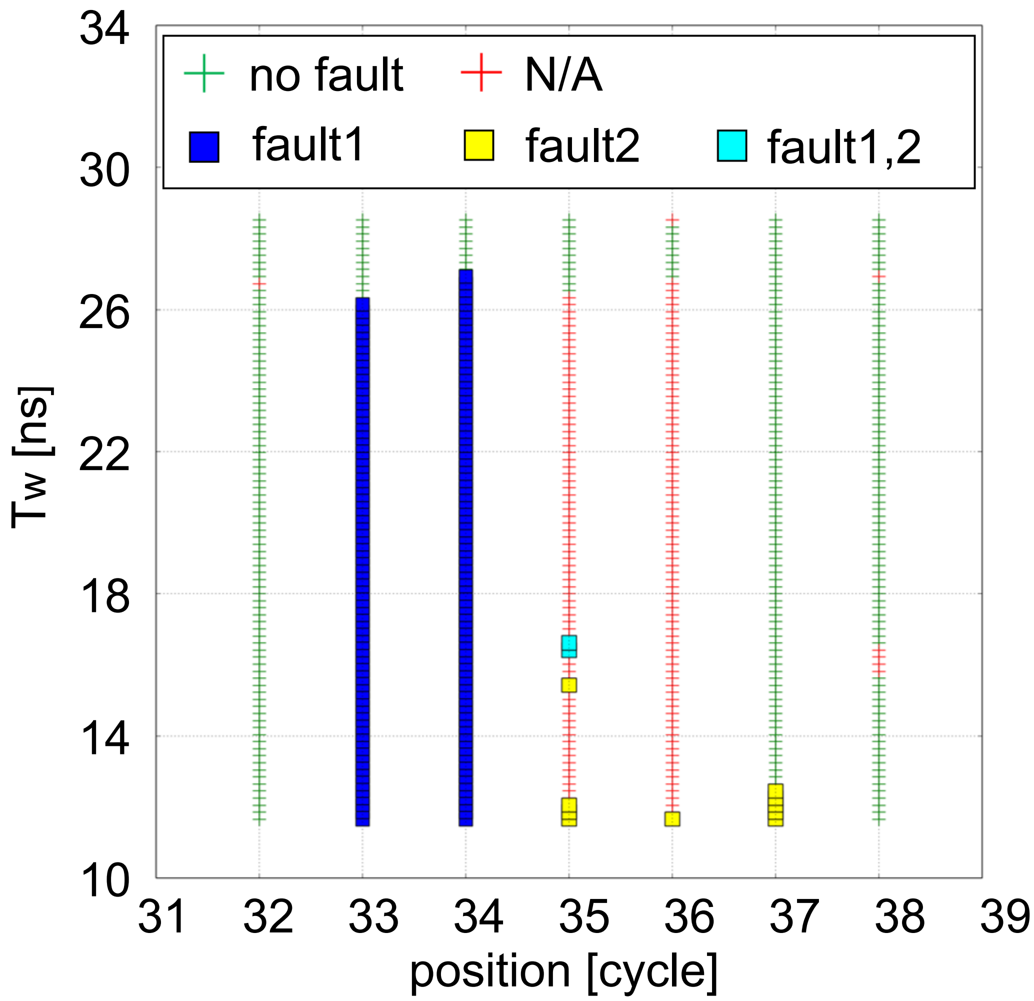


図 4.9　{ADDS, STRB} への故障感度解析

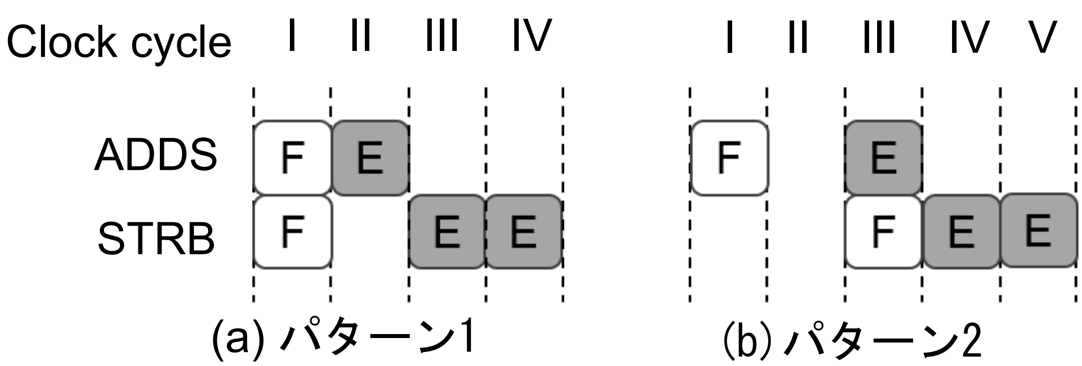


図 4.10　連続した {ADDS, STRB} のフェッチパターン

表 4.3　命令スキップのパターン

inst1 \ inst2	ADDS	MULS	STRB	LDRB	B	NOP
ADDS	[1]/ [1,2]	[1]/ [1,2]	[1]/ [1,2]	[1]/ [1,2]	[1]/ [1,2]	[1]
MULS	[1]/ [2]	[1]/ [1,2]	[1]/ [1,2]	[1]/ [1,2]	[1]/ [1,2]	[1]
STRB	[1,2]	[1], [2]/ [1,2]	[1], [2]	[1], [2]/ [1,2]	[1], [2]	[1]
LDRB	[1], [2]/ [1,2]	[1], [2]/ [1,2]	[2], [1,2]/ [1,2]	[1], [2]/ [2]	[1], [2]/ [1,2]	[1]
B	[1], [2]/ [1,2]	[1,2]	[1], [2]/ [1], [1,2]	[1], [2]/ [1,2]	[1]/ [1,2]	[1]

STRB (Fetch) (Tw $<$ 12[ns])
Ⅳ: STRB (Execute) (Tw $<$ 12[ns])
Ⅴ: STRB (Execute) (Tw $<$ 13[ns])
同様に，残りの 29 通りの命令グループに関しても実験を行った．なお，図 4.10 のどちらのフェッチパターンで実行されるかはプログラムに依存するため，解析対象の組合せの前に入れる NOP の数を調整することで，両パターンに対する命令スキップを観測できるように留意した．

故障感度解析の結果を表 4.3 にまとめる．ここでは，部分的にデータが誤る故障 ($\left[N/A\right]$) は無視し，命令スキップにのみ着目する．なお，NOP の列は単一の命令への故障注入を示す．表中に出現するスラッシュ"/" は，NOP の数を調整した 2 種類のプログラムにおいて，それぞれ別の命令スキップのパターンが現れたことを示す．例えば，先ほどの {ADDS, STRB} の場合，ADDS のみがスキップされる場合と，ADDS, STRB の両方がスキップされる場合があったことを意味する．図 4.9 においては，STRB のみがスキップされるパターンも見られたが，ここでは複数回の故障を入れることを念頭に置いて安定性を重視し，グリッチ幅を連続して 4[ns] 以上変更しても同じ命令スキップが観測された場合のみを表 4.3 に反映した．

攻撃にあたっては，3.4.2 節の方針に基づき，図 3.3 における (1) ループカウンタの確認 (CMP, BEQ)，および (2) ループカウンタの更新 (SUBS) をスキップする．それぞれの処理を攻撃対象とした場合の攻撃の実現可能性を表 4.4 にまとめる．ここでは，スキップ対象となる命令グループ候補と，各グループ候補への攻撃に要する命令スキップを合わせて示す．実現可能性に関しては次のように分類した．
△: グリッチ幅 (Tw) の調整により必要な命令スキップが得られる

表 4.4　`strncpy()` への攻撃可能性

攻撃対象	命令グループ	攻撃に要する命令スキップ	実現可能性
(1)	[Ⅰ] {CMP, BEQ}	[1], [2], [1,2]	◯
	[Ⅱ] {BEQ, LDRB}	[1]	△
(2)	[Ⅲ] {LDRB, SUBS}	[2], [1,2]	△
	[Ⅳ] {SUBS, STRB}	[1], [1,2]	◯

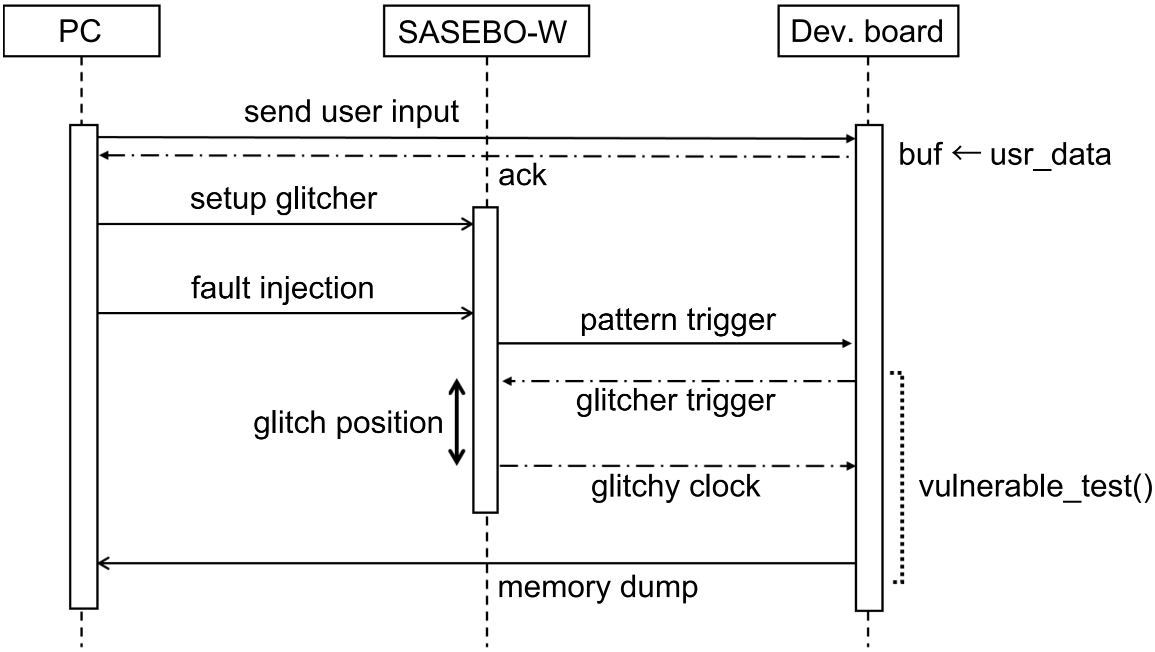


図 4.11　プログラムフロー (ARM)

◯: グリッチ幅 (Tw) の調整により得られる，どの命令スキップによっても攻撃が可能である

なお，`SUBS` は `ADDS` と動作がほぼ同じため `ADDS` と読み替え，`CMP` は演算結果を格納しない（フラグのみの更新を行う）`SUBS` であるため `ADDS` と読み替え，`BEQ` は同じ分岐命令である `B` に読み替えて表 4.3 を参照した．これらの命令を，データコピーのループ処理の中で繰り返しスキップすることで，入力サイズを任意の長さにまで増加させる．

4.3.2　**攻撃実験**

図 4.8 と同じ環境で行った実験結果を以下に示す．実験で使用したプログラムのフローは図 4.11 のようになる．まず，ARM マイコンでは，PC から送られた 32 バイトのユー

```
void vulnerable_test(void){
    uint8_t msg[10] = {0};

    strncpy(msg, buf, sizeof(msg) - 1);
    write_byte((uint8_t *)&msg[0], 32);
}
```

図 4.12　vulnerable_test() の C コード (ARM)

```
1 11 01 02 03   // msg[10]
2 04 05 06 07   // msg[10]
3 08 00 00 20   // msg[10]
4 00 04 01 40
5 27 05 00 08   // return address
6 00 00 00 00
7 00 04 00 50
8 20 00 00 00
```

図 4.13　スタックメモリ（故障なし）

ザ入力を 32 バイトのグローバル配列変数 buf へと格納し，故障注入タイミングの同期を取るために，無限ループ状態で待機する．次に，クロックグリッチ生成器のグリッチパラメータの設定を行うとともに，故障の注入準備を行う．SASEBO-W 上の FPGA から送信されたトリガ信号 (pattern trigger) により，無限ループ状態で待機していた ARM マイコンでは例外処理が発生し，同期信号 (glicher trigger) を FPGA へと送信すると共に，図 4.12 に示す vulnerable_test() を呼び出してデータ処理を行う．図 4.12 では，10 バイトの配列変数 msg が宣言されており，strncpy() を用いて最大で 9 バイト分だけ buf から msg へとコピーする．この時に，設定したグリッチパラメータに応じて故障が注入される．最後に，write_byte() という関数を用いて，msg[0] のアドレスから 32 バイトのメモリ内容をシリアル通信で PC へと送信する．このメモリダンプにより，スタックに格納されたリターンアドレス等の攻撃に必要な情報，および攻撃の成否を確認する．

まず，故障を注入せずに $0x110102...1e1f$ という連番 32 バイトのデータを送信した時のメモリダンプを図 4.13 に示す．連番の先頭が $0x00$ ではない理由は，strncpy() は NULL 文字 ($0x00$) が現れた時点でデータのコピーを終了するためである．strncpy() の仕様から，送信したデータのうち，$0x11$ から $0x08$ までの 9 バイトだけがストアされていることが分かる．また，実験で使用したプログラムのダンプから，5 行目の”27 05 00 08” がリターンアドレスであることが分かる．本実験では，このリターンアドレスを不正

```
00 00 00 00    // msg[10]
00 00 00 00    // msg[10]
00 09 0a 0b    // msg[10]
0c 0d 0e 0f
55 02 00 08    // modified return address
00 00 00 00
00 04 00 50
20 00 00 00
```

図 4.14　スタックメモリ（多重故障）

に書き換えることにより，任意の関数を呼び出す．ここでは，例として，ボード上の LED を点滅させる `blink_LED()` という関数 (アドレスは “08000255”) に書き換えることで，この関数を不正に呼び出す．アドレスはリトルエンディアンで格納されていることに注意し，攻撃コードは，“{ 任意の 16 バイト } + {0x55, 0x02} + {0 パディング }” の計 32 バイトとなる．また，現在のプログラムでは，入力サイズが 9 バイトに制限されているため，AVR マイコンを用いた実験と同様に多重故障を用いて，表 4.4 の命令を繰り返しスキップする．

多重故障注入を併用し，攻撃コードとして，任意の 16 バイトの部分に $0x110102...0f$ という連番を入れたものを送った時のメモリダンプを図 4.14 に示す．図 4.13 と比べて，配列変数 `msg` の最初の 9 バイトが $0x00$ に変わり，$0x09$〜$0x02$ までの 9 バイトが余計に読み込まれていることが確認できる．また，プログラムダンプによる確認と並行して，ボードの LED を観察することで `blink_LED()` の呼び出しに成功したことを確認した．この結果は，表 4.4 の Ⅳ の命令グループが $\left[1,2\right]$ の故障タイプでスキップされたことにより引き起こされたと推測できる．`SUBS` がスキップされることによりループカウンタの更新が行われず入力サイズが増加した一方，`STRB` もスキップされたことによりロードされた命令がストアされず，最初の 9 バイトは初期化された状態のままであったと考えられる．また，今回の実験では，表 4.4 の (1) に対しては攻撃が成功しなかった．このように，表 4.3 の結果通りの故障が得られなかった理由としては，`CMP` の挙動が `ADDS` とは異なることや，`CMP` の前に実行された無条件分岐 (`B`) の影響が考えられる．一方で，上記の実験結果から，故障注入攻撃と BOF 攻撃を併用することにより，ARM プロセッサ上においても汎用ソフトウェアに対する攻撃が可能となることが示された．

4.4　提案攻撃への対策

本節では，AVR マイコンと ARM マイコンのそれぞれのマイコンに対して，提案攻撃への対策を実装する．まず，提案攻撃がスキップ対象とする各命令に対して，命令スキップを防ぐための方針を立てる．次に，対策の方針に基づき，AVR マイコンと ARM マイコンのそれぞれについて，提案攻撃への対策を実装する．最後に，各マイコンにおいて生じる，対策によるオーバーヘッドの評価を行う．

4.4.1　対策の方針

提案攻撃への対策として，大きく 2 種類の対策を考えることができる．一つは BOF 攻撃対策，もう一つは故障注入攻撃対策である．BOF 攻撃対策は，表 2.3 に挙げたように，すでに多くの手法が開発されている．しかし，2.2.3 節で述べたように，対策の多くが高度な OS を搭載していることや十分な対策を実装できるだけのリソースを確保できることを前提としたものである．このため，マイコンの機種やアーキテクチャといった環境に依存せずに使用出来る対策としては，今回想定した入力データのサイズ制限が妥当であると言える．一方，故障注入攻撃は，2.3.4 節で述べたように，コストや対策の有効範囲といった面で，ソフトウェアでの対策が有効だと言える．そこで，本論文では，故障注入攻撃に対するソフトウェアでの対策を示す．なお，ここでは，対策の目的を BOF の発生を防ぐことにのみ絞り，スタックの内容が正常通りに保護されることは扱わない．これは，攻撃対象の組込み機器を異常停止する目的であれば，故障注入は必要なく，破壊行為を行えばよいためである．さらに，機器が異常に停止することは，BOF 攻撃により制御が奪取されることと比べると，影響度が低いと言えることも，理由として挙げられる．

第 3 章で述べたように，AVR と ARM のどちらのマイコンに対する攻撃に関しても，攻撃対象となる命令は二つある．一つは終了条件の判定を行う分岐命令であり，もう一つはループカウンタの更新命令である．分岐命令を守るためには，暗号の分野で故障注入攻撃への対策として考案された「デフォルトフェイル」が有効である [20], [52]．この手法では，プログラムフローは分岐命令以外では下位から上位へ遷移しないことを利用している．守りたい分岐命令を下位に配置することで，故障が入らずに正しく処理された場合にのみ，上位に存在する守りたい処理（鍵系列の出力等）を実行可能となる．本提案攻撃への対策として利用する場合には，ループ処理の最後に分岐命令を配置すればよい．これにより，分岐命令のスキップの有無にかかわらず，ループの継続条件を満たしていない場合には終了することになる．ただし，ループ内に二つ以上の分岐命令が存在する場合，この手法では一つの分岐命令しか守ることができない．従って，`strncpy()` の NULL チェッ

クに関しては，BOF 攻撃が発生した時の被害の大きさと比較すると影響度は低いと言えるため，ここでは妥協する．

また，ループカウンタの更新命令は，ループカウンタとロード（ストア）アドレスとを関連付けることで守ることが可能である（以降，この対策をカウンタとアドレスの結び付けと呼ぶ）．図 3.1, 3.3 におけるルーチンでは，ループカウンタとロード（ストア）アドレスが独立しているため，ループカウンタの更新をスキップすることにより，余計に入力サイズを増やすことが可能である．一方，ループカウンタとロード（ストア）アドレスを結びつけることにより，ループカウンタの更新がスキップされた場合には，スキップされる前のロード（ストア）アドレスから値が変化しないため，同じデータが同じアドレスへ格納されるだけであり，BOF は発生しない．具体的な実装としては，ロードやストア命令の実行時に，オフセットを用いた相対アドレッシングを用いる方法が挙げられる．図 3.1, 3.3 のコードでは，ロード（ストア）の際のシーケンシャルなデータアクセスの実現のために，ロード（ストア）アドレス自体をインクリメントして更新している．そこで，ループカウンタをインクリメントしていき，ループカウンタをオフセットとした相対アドレッシングを用いることで，この対策を実装できる．

4.4.2　AVR マイコンにおける対策

8 ビット AVR マイコンに本対策を実装する場合，デフォルトフェイルは 4.4.1 節の方針通りに実装可能である一方，カウンタとアドレスの結び付けに関してはそのまま実装することはできない．これは，8 ビット AVR マイコンでは，任意アドレスをオフセットとした相対アドレッシングが定義されていないためである．そこで，8 ビット AVR マイコンの場合は，ストアアドレスをループカウンタとしても扱うことにより，カウンタとアドレスの結び付けを適用する．これにより，ループカウンタ（ストアアドレス）の更新がスキップされた場合には，新たにロードされたデータは同じアドレスにストアされるため，入力サイズが増加したことにはならない．命令スキップにより，コピー後の値は高い確率で壊れることになるが，BOF が発生して任意の関数を呼び出されることと比較すると影響度は低いと言える．

対策を実装した `strncpy()` を `my_strncpy()` と呼び，図 4.15 に示す．ループカウンタのチェック後，データコピーを行うループ処理の先頭へ分岐する命令を 16, 24 行目のループの最後に配置することで，デフォルトフェイルを実現している．一方，カウンタとアドレスの結び付けは，複数の命令から実現される．まず，4, 5 行目の加算により，元のループカウンタの値 (R21:R20) にストアアドレス (X レジスタ, R27:R26) を足し合わせる．この値をループカウンタの最終値とし（これ以降，単に最終値と呼ぶ），以後，R21:R20 の代わりに，X レジスタをループカウンタとして扱う．元の `strncpy()` では，事後増加

```
1  my_strncpy:
2      MOVW R30, R22    ;  •Z レジスタに引数 1 をセット
3      MOVW R26, R24    ;  •X レジスタに引数 2 をセット
4      ADD  R20, R26    ;  •R21:R20 += R27:R26
5      ADC  R21, R27    ;   （カウンタの最終値）
6      RJMP CMP         ;  •カウンタサイズのチェックへ
7  LOOP:    ; メモリのデータコピーを行うループ
8      LD   R0, Z+      ;  •データをロード
9      ST   X, R0       ;  •データをストア
10     ADIW R26, 0x01   ;  •ループカウンタの加算
11     AND  R0, R0      ;  •NULL 文字（0x00）の
12     BREQ Z_CMP       ;    チェック
13 CMP:
14     CP   R26, R20    ;  •ループカウンタの
15     CPC  R27, R21    ;    チェック
16     BRLO LOOP        ;  •R27:R26 < R21:R20 でループ
17     RET
18 LOOP_Z:    ; 以下,余ったバッファの 0埋め処理
19     ST   X, R1
20     ADIW R26, 0x01
21 Z_CMP:
22     CP   R26, R20
23     CPC  R27, R21
24     BRLO LOOP_Z
25     RET
```

図 4.15　対策済み strncpy() (AVR)

付きのストア命令を用いていたが，my_strncpy() では，ST と ADIW (9, 10 行目) のようにそれぞれ分割して実行している．これは，自動でアドレスをインクリメントするストア命令を用いた場合，データコピーを行うループ処理の中に，キャリーフラグを変更する命令が 14, 15 行目の CP, CPC 命令しか無いため，この二つの比較命令を連続でスキップし続けることにより 16 行目の BRLO では常に分岐させることが可能であるためである．10 行目の ADIW はループカウンタ（ストアドレス）の更新であるため，この命令をスキップした場合には攻撃が成立しない．また，14, 15 行目の CP, CPC はコピー回数がサイズ制限に達したかどうかを確認する処理である．ストアアドレスから最終値を減算し，0 よりも小さい場合にはキャリーフラグが立ち，ループが継続される．このように，攻撃対象となる命令同士を関連付けることにより，命令スキップへの耐性を持たせることが可能となる．また，バッファの 0 埋め処理は，データのロードを行わないコピー処理であるため，基本的なルーチンは前半部分と同様である．

```
1  my_strncpy:
2      PUSH {R4, LR}
3      MOVS R3, #0         // ・ループカウンタのクリア
4      CMP  R3, R2         // ・ループカウンタの
5      BGE  3f             //   チェック(R3 >= R2 ?)
6  1:    // メモリのデータコピーを行うループ
7      LDRB R4, [R1, R3]   // ・データをロード
8      STRB R4, [R0, R3]   // ・データをストア
9      ADDS R3, #1         // ・ループカウンタ ++
10     CMP  R4, #0         // ・NULL 文字(0x00)の
11     BEQ  2f             //   チェック
12     CMP  R3, R2         // ・ループカウンタの
13     BLT  1b             //   チェック(R3 < R2 ?)
14 2:    // 以下,余ったバッファの 0埋め処理
15     CMP  R3, R2
16     BGE  3f
17     MOVS R1, #0
18     STRB R1, [R0, R3]
19     ADDS R3, #1
20     B    2b
21 3:
22     POP  {R4, PC}
```

図 4.16　対策済み strncpy() (ARM)

4.4.3　ARM マイコンにおける対策

Cortex-M0+ ベースのマイコンの場合，8 ビット AVR マイコンのようなアドレッシングの問題はないため，方針通りに対策を適用可能である．対策を施した strncpy() のコードを図 4.16 に示す．まず，デフォルトフェイルの適用のために，ループカウンタ (R3) のチェックを 12, 13 行目のループ処理の最後に配置した．次に，カウンタとアドレスの結びつけのために，7, 8, 18 行目のロード（ストア）命令をループカウンタの相対アドレッシングを用いるように変更した．また，これに伴い，元の strncpy() では，ループカウンタをデクリメントしていたのに対し，my_strncpy() では，ループカウンタはインクリメントするように変更した．なお，この変更に伴い，分岐の判定を，等価関係 (BEQ, BNE) から大小関係に変更した (BGE, BLT) (5, 13, 16 行目).

4.4.4　オーバーヘッド評価

strncpy() と my_strncpy() のプログラムサイズ，およびクロックサイクル数を，8

表 4.5　対策によるオーバーヘッド

		プログラムサイズ [byte]	クロックサイクル数 [cycle]
AVR	`strncpy()`	30	$10 + 10n$ $(m = 0)$ $20 + 10n + 6m$ $(m \geq 1)$
	`my_strncpy()`	40	$13 + 11n$ $(m = 0)$ $25 + 11n + 7m$ $(m \geq 1)$
ARM	`strncpy()`	38	$19 + 13n$ $(m = 0)$ $26 + 13n + 9m$ $(m \geq 1)$
	`my_strncpy()`	36	14 $(n = 0)$ $25 + 11n + 9m$ $(n \geq 1)$

※ n: 非 NULL 文字の数, m: NULL 文字の数

ビット AVR マイコンと Cortex-M0+ ベースの ARM マイコンそれぞれに関して表 4.5 に示す．クロックサイクル数の列における n はコピーする文字列中の先頭バイトからの非 NULL 文字の数であり，m は n に続く NULL 文字の数である．また，$n + m =$ `size` となる．例えば，“ABCDEF(0x00)(0x00)” という 8 バイトの文字列の場合，$n = 6, m = 2$ となる．

AVR マイコンでは，対策を施す場合，プログラムサイズは 10 バイトだけ増加する．この増加分 10 バイトは，実験で使用した ATmega163 のプログラムメモリのサイズ 16KB を考慮しても十分に小さいと考えられる．一方で，クロックサイクル数は，NULL 文字の有無にかかわらず，n や m，すなわち `size` に比例して増加する．データメモリの小さいマイコン（ATmega163 の場合は 1KB）においては，設定する入力サイズがあまり大きくならないことが予想できるため，入力サイズに比例したクロックサイクル数の増加は許容できる程度のオーバーヘッドだと言える．

ARM マイコンにおいては，対策を施した場合にはプログラムサイズが 2 バイトだけ減少する．一方，クロックサイクル数は，$m \geq 1$ の場合には n に比例して減少する．また，$m = 0$ の場合であっても n に比例して減少するため，$n \neq 1, 2$ であれば，対策済みの `my_strncpy()` の方がクロックサイクル数は小さくなる．これは，元の命令の構造に冗長な部分があることや，ARM で使用されている命令セットが高級であるため，AVR マイコンでの対策とは異なり，対策の方針通りに無理なく実装可能であったことが要因だと考えられる．

4.5 結び

本章では，第 3 章で提案した故障注入を併用したバッファオーバーフロー攻撃の有効性を実証し，同攻撃への対策手法を示した．まず，8 ビット AVR マイコンである ATmega163 を用いた実験を通して，提案攻撃の実現可能性を実証した．さらに，Cortex-M0+ ベースの 32 ビット ARM マイコンである STM32L053C8 を用いた実験を通して，提案攻撃が特定の機器やアーキテクチャに依存しない，幅広い機器に対して脅威となり得ることを実証した．その後，提案攻撃への対策の基本方針を述べ，各マイコンにおけるソフトウェアでの効果的な対策を示した．また，対策のオーバーヘッドについて考察した．

第 5 章

結言

以上，第 2 章から第 4 章まで，組込みソフトウェアに対する故障注入攻撃に関して，その実現可能性と攻撃対策について述べた．

第 2 章では，組込みソフトウェアへの攻撃に関する基礎的考察を行った．ソフトウェアへの論理攻撃として，幅広い組込み機器に対して有効である点と攻撃の影響の大きさから，バッファオーバーフロー (BOF) 攻撃に着目した．BOF 脆弱性は，データの入出力処理が存在するあらゆるソフトウェアに存在する可能性があること，また，BOF 脆弱性を利用した BOF 攻撃は，任意コードを実行できる可能性が高いため，危険度の高い攻撃であると言えることから，ローエンドな組込みソフトウェアに対する攻撃の中でも特に関心が高いと考えた．まず，具体例を用いて BOF 攻撃の原理を述べ，典型的な対策手法について述べた．その中でも，特に，機種やアーキテクチャの依存性が少なく，ローエンドなマイコンを使用した組込み機器でも利用可能である，入力サイズ制限が可能な関数を使用するという対策手法が最も一般的であることを述べた．その後，ハードウェアへの物理攻撃として故障注入攻撃に着目した．故障注入攻撃は，攻撃に要する金銭的コストや技術的知識が侵襲型の物理攻撃と比べて少ない一方，注入した故障により生じる命令スキップ等の現象はソフトウェアへの攻撃に有効であることから，組込みソフトウェアへの現実的な脅威となり得ると考えた．まず，故障注入手法を分類し，それぞれの特徴を述べた．次に，故障注入攻撃をソフトウェアへの攻撃に利用した既存研究について，それぞれの攻撃において使用されている故障注入手法と故障モデルを比較した．その後，こうした考察を通して，本実験に対して適切な故障注入手法を選択した．本論文では，時間的な分解能が高く，実験の再現性も高いクロックグリッチを故障注入手法として採用した．最後に，故障注入攻撃への対策について，フォールトトレランスと従来の故障注入攻撃への対策手法を比較し，ローエンドな機器における対策としては，故障注入攻撃に対するソフトウェアでの対策が実用的であることを述べた．

第 3 章では，故障注入を併用したバッファオーバーフロー攻撃について提案した．提案

攻撃のアイデアは，故障注入により生じるアセンブリレベルでの命令スキップを利用して，任意の命令を複数個スキップすることにより，ソフトウェアで提供される従来の攻撃対策を無効化することである．まず，故障注入攻撃と BOF に着目している点で本研究と類似する文献 [18] の研究と比較を行い，提案攻撃の優位点を述べた．次に，8 ビット AVR マイコンの例として，ATmega163 のアーキテクチャについて述べ，`strncpy()` を攻撃対象として，そのアセンブリ命令の解析を行い，提案攻撃を適用する方法について述べた．具体的には，こうしたコピー関数で用いられているループ構造に着目し，ループカウンタの更新命令，および入力制限サイズに到達した際に実行される分岐命令の 2 種類が命令スキップの対象となることを述べた．その後，ARM マイコンの中でも最もローエンド向けとして設計された Cortex-M0+ のアーキテクチャについて述べ，Cortex-M0+ ベースのマイコンに対して提案攻撃を適用する方法について述べた．ARM マイコン向けに実装された `strncpy()` のアセンブリ命令も，AVR マイコン向けのものと同様の構造となっており，スキップの対象となる命令の種類は変わらないことを述べた．

第 4 章では，第 3 章で提案した故障注入を併用したバッファオーバーフロー攻撃の有効性を実証し，同攻撃への対策手法を示した．まず，スマートカードへのサイドチャネル攻撃評価において広く使用されている 8 ビット AVR マイコン， ATmega163 を用いた実験について述べた．最初に，故障を注入せずにプログラムを動作させ，メモリの内容を調査することで，攻撃コードの作成を行うとともに，故障注入回数を決定した．その後，故障注入によりループカウンタの更新をスキップすることで入力サイズの制限を無効化し，通常の BOF 攻撃と同様に任意関数の呼び出しが可能であることを示した．次に，Cortex-M0+ ベースの 32 ビット ARM マイコンである STM32L053C8 を用いた実験について述べた．まず，故障感度解析実験を行い，攻撃の実現可能性について検討した．故障感度解析実験では，連続する 2 命令に対して生じる命令スキップのパターンを示した．次に，こうした解析結果から，提案攻撃の実現可能性を示し，攻撃実験を行った．AVR マイコンを用いた実験と同様の手順で攻撃を行い，任意関数を呼び出すことが可能であることを実証した．これにより，提案攻撃が特定の機種やアーキテクチャに依存せず，幅広い機器に対して適用可能であることを示した．最後に，提案攻撃への対策を検討し，各マイコンについて対策を施した場合に生じるオーバーヘッドの評価を行った．

今後の展望としては，提案攻撃の有効範囲を特定することが挙げられる．攻撃を適用するためには，多重故障注入攻撃を実行するための条件と BOF 攻撃を実行するための条件を満たすこと，さらに攻撃対象のプログラムが命令スキップに対する脆弱性を有することが必要となる．このうち，BOF 攻撃が実行可能である条件は既存研究から明らかになっているが，残りの二つに関しては調査の余地がある．多重故障注入攻撃の実行可能性については，多重に故障を注入するための手法は数多く存在することから，それぞれの手法の限界を評価することは極めて困難である．攻撃対象のプログラムが命令スキップに対する

脆弱性を有するかどうかに関しては，第 3 章で考察したように，(1) ループ構造を有すること，(2) ユーザ入力がメモリからロードされること，(3) そのデータがメモリへとストアされること，という三つの条件を満たすかどうかを調べれば良い．現状では，手動でアセンブリコードを解析して提案攻撃への脆弱性を有するかどうかを確認しているため，脆弱性を検出するための自動化手法が必要だと考えられる．併せて，提案攻撃への対策の適用，およびその有効性の証明に関しても，自動化手法が求められる．

参考文献

[1] IPA 独立行政法人 情報処理推進機構, “IT 人材白書 2015.” `http://www.ipa.go.jp/files/000045391.pdf`, April 2015.

[2] Government Office for Science - GOV.UK, “The internet of things: making the most of the second digital revolution.” https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/409774/14-1230-internet-of-things-review.pdf, 2014.

[3] Heffner Craig, “Exploiting network surveillance cameras like a hollywood hacker,” in *Black Hat USA 2013*, August 2013.

[4] Miller Charlie and Valasek Chris, “Remote exploitation of an unaltered passenger vehicle,” in *Black Hat USA 2015*, August 2015.

[5] Lee SeungJin, B́eist, “Hacking, surveilling, and deceiving victims on smart tv,” in *Black Hat USA 2013*, August 2013.

[6] Grattafiori Aaron and Yavor Josh, “The outer limits: Hacking the samsung smart tv,” in *Black Hat USA 2013*, August 2013.

[7] 安藤繁, 田村陽介, 戸辺義人, 南正輝, センサネットワーク技術：ユビキタス情報環境の構築に向けて. 東京電機大学出版局, May 2005.

[8] Thomas Krak and Michael Hoefler, “On the effects of clock and power supply tampering on two microcontroller platforms,” in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014 Workshop on*, pp. 8–17, IEEE, 2014.

[9] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz, “Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller,” in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pp. 77–88, IEEE, 2013.

[10] Michel Agoyan, Jean-Max Dutertre, Amir-Pasha Mirbaha, David Naccache, Anne-Lise Ribotta, and Assia Tria, “How to flip a bit?,” in *Proceedings of the 2010 IEEE 16th International On-Line Testing Symposium*, pp. 235–239, IEEE Computer Society, 2010.

[11] Dan Boneh, Richard A DeMillo, and Richard J Lipton, "On the importance of checking cryptographic protocols for faults," in *Advances in Cryptology-EUROCRYPT' 97*, pp. 37–51, Springer, 1997.

[12] Pierre-Alain Fouque, Reynald Lercier, Denis Réal, and Frédéric Valette, "Fault attack on elliptic curve montgomery ladder implementation," in *Fault Diagnosis and Tolerance in Cryptography, 2008. FDTC'08. 5th Workshop on*, pp. 92–98, IEEE, 2008.

[13] Eli Biham and Adi Shamir, "Differential fault analysis of secret key cryptosystems," in *Advances in Cryptology-CRYPTO'97*, pp. 513–525, Springer, 1997.

[14] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo, "Differential fault analysis on aes," in *Applied Cryptography and Network Security*, pp. 293–306, Springer, 2003.

[15] Sudhakar Govindavajhala and Andrew W Appel, "Using memory errors to attack a virtual machine," in *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pp. 154–165, IEEE, 2003.

[16] Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin, "Attacks on java card 3.0 combining fault and logical attacks," in *Smart Card Research and Advanced Application*, pp. 148–163, Springer, 2010.

[17] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet, "Combined software and hardware attacks on the java card control flow," in *Smart Card Research and Advanced Applications*, pp. 283–296, Springer, 2011.

[18] Pierre-Alain Fouque, Delphine Leresteux, and Frédéric Valette, "Using faults for buffer overflow effects," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pp. 1638–1639, ACM, 2012.

[19] Elena Trichina and Roman Korkikyan, "Multi fault laser attacks on protected crt-rsa," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*, pp. 75–86, IEEE, 2010.

[20] Sho Endo, Naofumi Homma, Yu-ichi Hayashi, Junko Takahashi, Hitoshi Fuji, and Takafumi Aoki, "A multiple-fault injection attack by adaptive timing control under black-box conditions and a countermeasure," in *Constructive Side-Channel Analysis and Secure Design*, pp. 214–228, Springer, 2014.

[21] Yves Younan Sourcefire Vulnerability Research Team, "25 years of vulnerabilities: 1988-2012." `https://courses.cs.washington.edu/courses/cse484/14au/reading/25-years-vulnerabilities.pdf`.

[22] 松本勉, 大石和臣, 高橋芳夫, "実装攻撃に対抗する耐タンパー技術の動向," 情報処理

学会論文誌, Vol. 49, No. 7, pp. 799–809, July 2008.

[23] N Paulauskas and E Garsva, "Computer system attack classification," *Elektronika ir Elektrotechnika*, Vol. 66, No. 2, pp. 84–87, 2015.

[24] The MITRE Corporation, "CWE: Common Weakness Enumeration." http://cwe.mitre.org/index.html.

[25] Inc First.org, "CVSS: Common Vulnerability Scoring System." https://www.first.org/cvss.

[26] NIST: National Institute of Standards and Technology, "NVD: National Vulnerability Databse." https://nvd.nist.gov/cwe.cfm.

[27] JPCERT/CC and IPA, "JVN iPedia: 脆弱性対策情報データベース." http://jvndb.jvn.jp/.

[28] C Foster James, Osipov Vitaly, Bhalla Nish, and Heinen Niels, "Buffer overflow attacks: Detect, exploit, prevent," 2005.

[29] 上原孝之, 情報処理教科書 情報セキュリティスペシャリスト 2013 年版. 翔泳社, 2012.

[30] IPA 独立行政法人 情報処理推進機構, "バッファオーバーフロー: #5運用環境における防御." `https://www.ipa.go.jp/security/awareness/vendor/programmingv2/contents/c905.html`, 2007.

[31] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, pp. 298–307, ACM, 2004.

[32] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.," in *Usenix Security*, vol. 98, pp. 63–78, 1998.

[33] S Andersen and V Abella, "Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies," 2004.

[34] Ingo Molnar, "Exec shield," *new Linux security feature*, 2003.

[35] CERT Coordination Center, "STR03-C. Do not inadvertently truncate a string." `https://www.securecoding.cert.org/confluence/display/c/STR03-C.+Do+not+inadvertently+truncate+a+string`.

[36] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermea-

sures," *Proceedings of the IEEE*, Vol. 100, No. 11, pp. 3056–3076, 2012.

[37] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, Vol. 94, No. 2, pp. 370–382, 2006.

[38] Jörn-Marc Schmidt and Christoph Herbst, "A practical fault attack on square and multiply," in *Fault Diagnosis and Tolerance in Cryptography, 2008. FDTC'08. 5th Workshop on*, pp. 53–58, IEEE, 2008.

[39] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria, "Electromagnetic transient faults injection on a hardware and a software implementations of aes," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2012 Workshop on*, pp. 7–15, IEEE, 2012.

[40] Sergei P Skorobogatov and Ross J Anderson, "Optical fault induction attacks," in *Cryptographic Hardware and Embedded Systems-CHES 2002*, pp. 2–12, Springer, 2003.

[41] Chong Hee Kim and Jean-Jacques Quisquater, "Fault attacks for crt based rsa: New attacks, new results, and new countermeasures," in *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems*, pp. 215–228, Springer, 2007.

[42] ENDO Sho, Takeshi Sugawara, Naofumi Homma, and Akashi SATOH, "A configurable on-chip glitchy-clock generator for fault injection experiments," *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. 95, No. 1, pp. 263–266, 2012.

[43] 南谷崇, フォールトトレラントコンピュータ. オーム社, 1991.

[44] Parag K. Lala, *Fault Tolerant & Fault Testable Hardware Design.* Prentice-Hall International (UK) Ltd, 1985.

[45] Mathieu Ciet and Marc Joye, "Practical fault countermeasures for chinese remaindering based rsa," *FDTC*, Vol. 5, pp. 124–132, 2005.

[46] Sikhar Patranabis, Abhishek Chakraborty, and Debdeep Mukhopadhyay, "Fault tolerant infective countermeasure for aes." Cryptology ePrint Archive, 2015.

[47] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni, "Countermeasures against fault attacks on software implemented aes: effectiveness and cost," in *Proceedings of the 5th Workshop on Embedded Systems Security*, p. 7, ACM, 2010.

[48] AIST, "Evaluation environment for side-channel attacks." `http://www.risec.aist.go.jp/project/sasebo/`.

[49] Atmel Corporation, “ATmega163(L).” `http://www.atmel.com/images/doc1142.pdf`.

[50] Yiu Joseph, *The Definitive Guide to ARM©Cortex©-M0+ and Cortex-M0+ Processors: Second Edition*. Newnes, 2015.

[51] STMicroelectronics, “32l0538discovery.” `http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/LN1848/PF260319?sc=stm32l0-discovery`.

[52] Marc Witteman and Martijn Oostdijk, “Secure application programming in the presence of side channel attacks,” in *RSA conference*, vol. 2008, 2008.

謝辞

本論文は，著者が東北大学 大学院情報科学研究科 情報基礎科学専攻 計算機構論分野 (青木 (孝)・本間 (尚) 研究室) において行った研究を取りまとめたものです.

本研究を推し進めるにあたり，恩師青木孝文教授には，筆者が学部 4 年次の分野配属以来，終始熱心な御指導と御鞭撻を頂きました．先生の研究・教育に対する真摯な御姿勢から多くを学んだことを銘記し，ここに改めて感謝の意を表します.

本論文をまとめるにあたり，本論文の審査員として御専門の立場から有意義な御意見・御助言を頂いた亀山充隆教授ならびに菅沼拓夫教授に深く感謝の意を表します.

本間尚文准教授には，研究に対する懇切なる御指導と終始変わらぬ励ましを頂くとともに，本論文の執筆においても様々な御助言を賜りました．ここに改めて感謝の意を表します.

最後に，日頃の研究室生活において様々な面で御協力頂いた伊藤康一助教をはじめとする研究室諸氏に心より御礼申し上げます.

2016 年 2 月 10 日