Formal Semantics Extraction from Natural Language Specifications for ARM

Anh V. Vu and Mizuhito Ogawa

Japan Advanced Institute of Science and Technology
{anhvv,mizuhito}@jaist.ac.jp

Abstract. This paper proposes a method to systematically extract the formal semantics of ARM instructions from their natural language specifications. Although ARM is based on RISC architecture and the number of instructions is relatively small, an abundance of variations diversely exist under various series including Cortex-A, Cortex-M, and Cortex-R. Thus, the semi-automatic semantics formalisation of rather simple instructions results in reducing tedious human efforts for tool developments e.g., the symbolic execution. We concentrate on six variations: M0, M0+, M3, M4, M7, and M33 of ARM Cortex-M series, aiming at covering IoT malware. Our systematic approach consists of the semantics interpretation by applying translation rules, augmented by the sentences similarity analysis to recognise the modification of flags. Among 1039 collected specifications, the formal semantics of 662 instructions have been successfully extracted by using only 228 manually prepared rules. They are utilised afterwards to preliminarily build a dynamic symbolic execution tool for Cortex-M called Corana. We experimentally observe that Corana is capable of effectively tracing IoT malware under the presence of obfuscation techniques like indirect jumps, as well as correctly detecting dead conditional branches, which are regarded as opaque predicates.

Keywords: Semantics Formalisation · Dynamic Symbolic Execution · IoT Malware Analysis · Natural Language Processing · ARM Cortex-M.

1 Introduction

Symbolic execution [1] is an old, powerful, and popular technique to analyse and/or verify software. It has been developed mainly for high-level programming languages, such as C and Java. Recently, the number of symbolic execution tools for binaries has gradually increased (e.g., Mcveto [2], Miasm [3], Mayhem [4], Klee-mc [5], Codisasm [6], Be-pum [7], and Angr [8]); however, most of them target x86 architecture. When analysing the dynamic behaviour of malware, the major obstacles are obfuscated codes (e.g., indirect jumps, opaque predicates, self-modification), which can be effectively solved by applying dynamic symbolic execution (also known as concolic testing). In particular, the concolic testing is able to dynamically explore the hidden destination of indirect jumps, whilst the symbolic execution can discover dead conditional branches, which will be eventually ignored. Considering the evolving threats of IoT malware, extending such tools to disparate architectures (e.g., ARM, MIPS, and PowerPC) becomes highly desired. There are two existing approaches to interpreting machine codes:

- Translating to an intermediate representation (e.g., LLVM in Klee-mc and VEX in Angr), where the coverage performance basically depends on the translators, such as Valgrind [9] in Klee-mc and Capstone [10] in Angr.
- Interpreting directly from binary codes, such as Mcveto and Be-pum (x86).

When the obfuscations exist, the former shares the difficulties with syntax-based disassemblers (e.g., Ida [11] and Capstone), which typically fail to disassemble malware [12]. We adopt the latter approach, which is more powerful; however, since it heavily requires a platform-wise implementation, an expensive engineering effort must be paid (e.g., 3155 instructions for x86-64 are counted in [13]). Contrary to a general impression on the intricacy of binaries, the good news is:

- IoT malware is mainly an user-mode sequential program without floating-point arithmetic. Avoiding multi-threads, weak memory models, and floating-point arithmetic allows us to consider a simple semantics framework as the transitions on the environment made by *memory*, *stack*, *registers*, and *flags*.
- Each instruction set officially contains a rigid natural language specification.
- Since various debuggers and emulation environments are available, the ambiguity occurring in the natural language processing can be resolved by testing.

This intuitively suggests the feasibility of semi-automatically formalising the semantics of rather simple instructions from their natural language specifications.

ARM is based on RISC architecture, thus, it has relatively few instructions ($\simeq$ 60 - 300). However, various series diversely exist such as Cortex-A for rich operating systems (e.g., Android OS), Cortex-R for real-time systems (e.g., LTE modems), and Cortex-M for micro-controllers (e.g., IoT devices). Moreover, each of them has numerous variations (e.g., 16 in Cortex-A, 5 in Cortex-R, and 9 in Cortex-M), which have been steadily increasing. Our study intentionally focuses on ARM Cortex-M, aiming at covering IoT malware. After collecting the official specifications of Cortex-M instructions on ARM developer website [14], their formal semantics are extracted by a systematic method, and the obtained semantics are utilised afterwards to preliminarily develop a dynamic symbolic execution tool for Cortex-M called Corana (Cortex Analyser) [15]. Note that, instead of trying to provide a fully automatic approach, our ultimate goal is significantly reducing tedious human efforts by automatically handling rather simple but many instructions, thus enables human to mainly concentrate on most complex parts.

Extraction Overview. Fig. 1 briefly illustrates an overview of the semantics extraction, where manually prepared tasks are bounded with dashed boxes. For each instruction i, among 5 sections from its natural language specification (section 2.1), 3 sections are utilised: *syntax* (name and arguments of i), *operation* (an informal interpretation of i), and *flags-update* (describing whether flags are modified after i is executed). Given a sentence S, after normalising its syntax (section 3) (I), if S comes from the *operation* section, the semantics interpretation (II) based on rewriting rules translates the normalised syntax tree to a Java code statement (section 4). If S is from the *flags-update* section, the similarity analysis (III) recognises whether the flags are modified (section 5). Thereafter, a Java method is automatically generated by instantiating the interpreted data into a pre-defined template, which represents the semantics framework as a transition on the environment. The correctness of generated methods is then verified using a conformance testing by comparing the execution results with a trusted emulator (section 6). By instantiating the extracted semantics into a prepared framework, CORANA is created (section 7). Our experiments on the sampled IoT malware reported in section 8 show that CORANA is capable of dynamically handling conditional data instructions and indirect jumps, as well as detecting dead branches, which are regarded as typical obfuscation techniques in IoT malware.

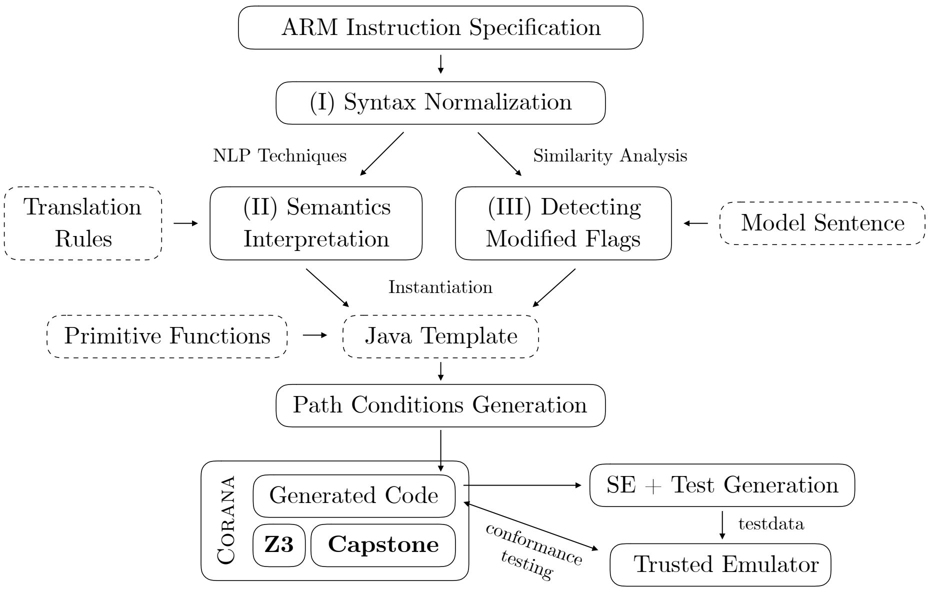

Fig. 1. A high-level overview of our semantics extraction approach

Related Work. There are several works focusing on extracting the specification from natural language descriptions. Nevertheless, they are mostly for human understanding (e.g., the requirements [16] and UML [17]), rather than the formal semantics of binaries. An interesting approach for the synthesis of x86-64 semantics is by learning formulas on BitVectors [18]. They confirmed the correctness by random testing, in which the results between their STRATA [18] and STOKE [19] are compared. Alternatively, an expensive human effort must be paid to describe the formal semantics, such as 3155 x86-64 instructions in the K-framework [13].

In fact, the formal semantics implicitly appears in the implementation of numerous binary emulators (e.g., μVision [20]) and symbolic execution tools (e.g., MCVETO [2], MIASM [3], MAYHEM [4], KLEE-MC [5], CODISASM [6], BE-PUM [7], and ANGR [8]). Whilst MIASM, MAYHEM, ANGR, and KLEE-MC first translate machine codes into an intermediate representation, Mcveto, Codisasm, and Be-pum directly interpret x86 binaries. Except for Mcveto and Miasm, they support the dynamic symbolic execution. Be-pum would be the first study of applying the binary semantics extraction from the natural language specifications [21]. After a three-year effort of the manual implementation, Be-pum roughly supported 250 instructions. Thereafter, the automatic extraction successfully generated 299 among 530 collected specifications, and 5 semantics bugs in the manual implementation were reported. At the moment, Be-pum covers around 400 instructions in total. Since the pseudocodes of x86 instructions are explicitly included in the Intel Developer Manuals, the semantics extraction was pretty simple by preparing roughly 30 primitive functions appearing in the pseudocodes. In contrast, the specifications of ARM instructions are given entirely in natural language, which makes the formalisation process [22] become more challenging.

2 Formal Semantics of ARM

2.1 Natural Language Specification

The specification of a Cortex-M instruction collected from the official ARM developer website [14] consists of five sections: *mnemonic*, *description*, *syntax*, *operation*, and *flags-update*. Table 1 shows an example of the rigid natural language specification (given in English) of the instruction UMAAL in ARM Cortex-M7.

Table 1. The natural language specification of UMAAL in ARM Cortex-M7

Mnemonic	UMAAL
Description	Signed multiply with accumulate long
Syntax	UMAAL{cond} RdLo, RdHi, Rn, Rm
Operation	The UMAAL instruction multiplies the two unsigned 32-bit integers in the first and second operands. Adds the unsigned 32-bit integer in RdHi to the 64-bit result of the multiplication. Adds the unsigned 32-bit integer in RdLo to the 64-bit result of the addition. Writes the top 32-bits of the result to RdHi. Writes the lower 32-bits of the result to RdLo.
Flags-Update	This instruction does not affect the condition code flags.

2.2 Operational Semantics

The implementation of numerous binary analysis tools (e.g., binary emulators, binary symbolic execution engines) implicitly contains the formal semantics of instructions, which have been formally defined in several recent studies (e.g., for x86 [23,24,13]). Although the semantics of binaries is seemingly intricate for human, the semantics framework for sequential programs is rather simple, which rigidly consists of a tuple of four ingredients: *registers*, *flags*, *memory*, and *stack*.

Definition 1. *The environment model* $E = \langle F, R, M, S \rangle$ *of the 32-bit ARM Cortex-M binaries consists of:*

- *F: a set of 6 flags:* $F = \{N, Z, C, V, Q, GE\}$
- *R: a set of 17 registers:*

$$R = \{r_0, r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}, r_{11}, r_{12}, sp, lr, pc, apsr\}$$

  where apsr is a special register storing the values of all flags N, Z, C, V *(also includes* Q, GE *in some particular versions of ARM).*
- *M: a set of n contiguous memory locations:* $M = \{m_0, m_1, \ldots, m_{n-1}\}$
- $S(\subseteq M)$*: a set of k contiguously allocated memory to store the stack:* $S = \{s_0, s_1, \ldots, s_{k-1}\}$ *with* $k < n$.

Since our target (IoT malware) is mainly a sequential user-mode process, the weak memory models and multi-threads are omitted. Accordingly, the execution of an instruction i is simply regarded as a transition t_i on a quadruplet in Fig. 2:

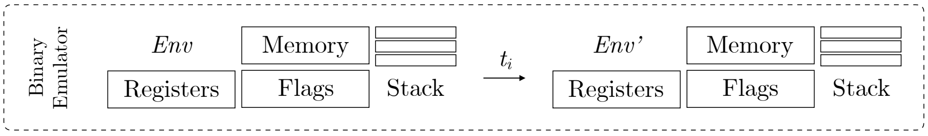

Fig. 2. The semantics transition t_i while executing an instruction i

For instance, the formal semantics of UMAAL is described in the SOS style [25]:

$$\frac{R_{pc} = k; instr(k) = umaal\ rdlo\ rdhi\ rn\ rm; R_{rdlo} = lo; R_{rdhi} = hi; \quad R_{rn} = n; R_{rm} = m; a = m * n + lo + hi; hi' = a \gg 32; lo' = (a \ll 32) \gg 32;}{\langle F, R, M, S\rangle \rightarrow \langle F, R[pc \leftarrow k + \mid instr(k) \mid; R_{rdlo} \leftarrow lo'; R_{rdhi} \leftarrow hi'], M, S\rangle} \text{[UMAAL]}$$

2.3 Java Specification as Formal Semantics

The formal semantics of an instruction i, which is considered as a transition t_i on the environment, is represented by a Java specification built on top of a customised class `BitVec` – a pair $\langle bs, s\rangle$ where bs is a `BitSet` value (a 32-bit BitVector supported in Java by default) and s is a string. After executing i, the concrete result of the operators is stored in bs whilst the corresponding symbolic formula (in SMT format of BitVector theory) is represented by s (an example is shown in section 7.1). In particular, this formal specification is technically obtained by instantiating the missing parameters into a pre-defined Java template:

```
public void $name($params, Character suffix) {
   arithmeticMode = $arithmeticMode;
   char[] flags = new char[]{$flags};
   BitVec result = null;
   $execCode
   if (suffix != null && suffix == 's') {
      if (result != null) {
         updateFlags(flags, result);
      }
   }
}
```

where:

1. The parameters that need to be instantiated: `params` (the missing arguments of this method), `name` (the instruction name), `arithmeticMode` (to specify whether the floating-point arithmetic is required), `execCode` (the main formal interpreted operations), and `flags` (the list of flags that might be modified).
2. Default arguments: `suffix` (if the suffix `s` occurs, the flags appearing in `flags` might be optionally updated based on the result of operators in `execCode`).
3. Manually prepared methods: `updateFlags` (update flags occurring in `flags`).

For instance, the generated Java method representing the semantics of UMAAL is described as follows, where dashed boxes indicate the instantiated parameters:

```
public void UMAAL(Character l, Character h, Character n,
    Character m, Character suffix) {
    arithmeticMode = ArithmeticMode.BINARY;                      syntax
    char[] flags = new char[]{};          flags-update
    BitVec result = null;
    result = mul(val(n),val(m));
    result = add(result,val(h));
    result = add(result,val(l));
    write(h,shift(result,Mode.RIGHT,32));
    write(l,shift(shift(result,Mode.LEFT,32),Mode.RIGHT,32));
    if (suffix != null && suffix == 's') {
       if (result != null) {                                 operation
          updateFlags(flags, result);
       }
    }
}
```

To interpret the Java specifications, 35 simple primitive functions are manually prepared, including arithmetic operators (e.g., `add`, `sub`, `mul`), logical operators (e.g., `and`, `or`, `xor`), and IO operators (e.g., `write`, `load`, `store`). Note that, some pre-defined functions do not have any corresponding representations in SMT format by default, thus their macros must be additionally declared (e.g., `bvmin`, `bvmax`, `bvabs`, `bvclz`). Among them, some instructions especially contain loops, which must be unfolded to be acceptable by theorem provers. A representative instance is the `clz r` instruction, which aims at counting the number of leading zeros of the value stored in the register `r`. Whilst its standard implementations normally require executing a loop, considering a 32-bit architecture, its macro in SMT format can be unfolded by iterating up to 32 times as indicated below:

```
(declare−const r0 (_ BitVec 32))
(declare−const c0 (_ BitVec 32))
... same declarations for r1,c1 ... r31,c31 ...
(declare−const r32 (_ BitVec 32))
(declare−const c32 (_ BitVec 32))
(declare−const z (_ BitVec 32))
(declare−const m (_ BitVec 32))
(define−fun clz ((x (_ BitVec 32))) (_ BitVec 32)
(if (and
  (= r0 x) (= z #x00000000) (= m #x00000001) (= c0 #x00000020)
  (= c1 (ite (bvsgt (bvashr r0 m) z) (bvsub c0 m) c0))
  (= r1 (ite (bvsgt (bvashr r0 m) z) (bvashr r0 m) r0))
  ... same declarations for c2,r2 ... c31,r31 ...
  (= c32 (ite (bvsgt (bvashr r31 m) z) (bvsub c31 m) c31))
  (= r32 (ite (bvsgt (bvashr r31 m) z) (bvashr r31 m) r31))
) c32 #x00000021))
```

3 Syntax Normalisation

Before proceeding further analyses, each raw sentence in the *operation* section is sequentially normalised by *parsing*, *lemmatisation*, and *words refinement*. In the implementation, we utilise *parsing* and *lemmatisation* modules provided in an open library namely NLTK [26]. Fig. 3 illustrates an example of the normalisation applied on the first sentence in the *operation* section of the *UMAAL* specification: *S – "The UMAAL instruction multiplies the two unsigned 32-bit integers in the first and second operands"*, which contains three sequential steps.

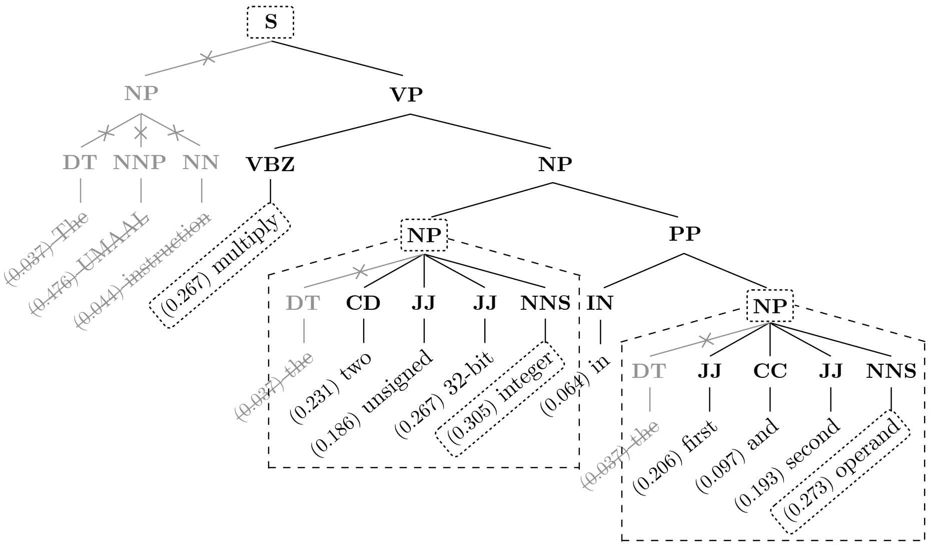

Fig. 3. The syntax tree, lemmatisation results, and TF·IDF of words in *S*

(1) Parsing. Parsing is applied for transforming each sentence to its structured syntax tree along with the corresponding labels in the grammatical categories (e.g., NP – Noun Phrase, DT – Determiner) based on the context-free grammar.

(2) Lemmatisation. Words written in English might have various expressions, such as conjugations and plural forms. Lemmatisation aims at unifying them to their standardised state. For instance, in Fig. 3, the words bounded with dashed boxes at the leaves of the syntax tree are the normalised results of this lemmatisation: *multiplies* → *multiply*, *integers* → *integer*, and *operands* → *operand*.

(3) Words Refinement. The popular measure TF·IDF [27] is utilised to effectively refine unimportant words in each sentence. For instance, the TF·IDF of words in S are put along with the leaves of the syntax tree. By setting a threshold $h = 0.05$, the strikethrough words in Fig. 3 are deleted and the removal recursively propagates to the root. Note that, the instruction name is also removed.

4 Semantics Interpretation by Translation Rules

Our semantics interpretation adopts a *rule-based* approach, which utilises the normalised syntax tree of sentences as the input. The key intuitive idea is: a *less* number of manually prepared rules can cover a *large* number of instructions. Firstly, we extract some popular phrases from the normalised syntax trees, which we named *NP-Phrases* (section 4.1). Thereafter, a set of appropriate instructions is carefully selected to obtain an *optimal trade-off* – the ratio between the number of rules needed and the number of covered instructions (section 4.2). Next, the translation rules are manually described by a recursive process (section 4.3). Eventually, by employing these prepared rewriting rules, the formal semantics of instructions are interpreted in a *bottom-up* manner (section 4.4). Note that, since our method is *sentence-wise*, the interpretation proceeds in sequence. Therefore, if an operation description is constituted of multiple sentences, the actual order of generated Java statements rigidly corresponds to the order of these sentences.

4.1 NP-Phrases Extraction

As a result of the syntax normalisation in section 3, all unimportant terms in the syntax trees are finally removed. We now extract some particular phrases by concentrating on sub-trees with the root label "NP". An *NP-Term* is a flattened string of a sub-tree in the normalised syntax tree in which only the root is labelled "NP". An *NP-Phrase* is either an NP-Term or the flattened string of the whole normalised syntax tree after substituting each NP-Term by an indexed blank hole $\square_i$. For instance, in Fig. 3, the sub-trees surrounded by dashed lines are NP-Terms, and the extracted NP-Phrases are: *"two unsigned 32-bit integer"*, *"first and second operand"*, and *"multiply* $\square_2$ *in* $\square_1$*"*. These NP-Phrases are further utilised as the input of the instruction selection strategy described in section 4.2.

4.2 Instructions Selection Strategy

We observe: (1) an instruction may carry various semantics in different variations (e.g., the instruction UASX appears both in M33 and M4, but the *flag-updates* sections are slightly different), and (2) since some instructions are presented by long and complex descriptions though appearing only once among all variations, they do not pay off the effort for preparing the corresponding rules (e.g., STLEX, VLLDM, and LDAEX only appear in M33). Thus, we aim to seek a set of appropriate candidates to obtain an *optimal trade-off*. The very high-level strategy is:

The importance of an instruction i is measured by the sum of TF·IDF scores of NP-Phrases in i. Select k instructions that maximise the sum divided by k.

To be more specific, for a list of k chosen candidates, we use $\varphi(k)$ to measure the efficiency of the selection strategy over all instructions in six variations. The greater $\varphi(k)$, the better selected candidates. Let I is the set of all n instructions:

$$I = \{i_1, i_2, \ldots, i_n\}$$

where an instruction i consists of a set T_i including w NP-Terms:

$$T_i = \{\langle t_1, f_1\rangle, \langle t_2, f_2\rangle, \ldots, \langle t_w, f_w\rangle\}$$

where t_j is the j^{th} NP-Term, and f_j is the frequency of t_j in i. Let $p(t_j)$ is the proportional occurrence of t_j over all NP-Terms in I, the importance of i over I is defined as:

$$m_i = \sum_{j=1}^{w} p(t_j).f_j$$

Let M is the sorted set (descending) of all m:

$$M = sorted(m_1, m_2, \ldots, m_{n-1}, m_n)$$

Let M_q is the q^{th} value of M, k is the number of expected candidates, $\varphi(k)$ is then defined as:

$$\varphi(k) = \frac{1}{k}\sum_{q=1}^{k} M_q$$

Now, given k, this strategy can effectively obtain an optimal trade-off by taking the first k candidates in M to make $\varphi(k)$ as large as possible. Obeying to this strategy, 692 instructions are selected. After combining similar NP-Phrases as conditional terms, 228 selected NP-Phrases become a set of left-hand side (LHS) candidates, which is further utilised as the input of the rules preparation process.

4.3 Translation Rules Preparation

A semantics interpretation rule translates a left-hand side (LHS) – an NP-Phrase, to a right-hand side (RHS) – a Java code statement. Note that, the LHS candidates, which are automatically selected by the strategy presented in section 4.2, are classified into 2 categories: NP-Phrase LHS (e.g.,*"first and second operand"*) and Context-Based LHS (e.g.,*"multiply* $\square_2$ *in* $\square_1$*"*). Additionally, a conditional LHS can be used to combine LHSes carrying similar semantics (e.g., *"halfword data"* and *"halfword value"* : $\langle$*halfword data* | *halfword value*$\rangle$). The RHSes are systematically prepared by a flow depicted in Fig. 4, including 5 following steps:

1. The set of LHS candidates (C) is sorted (descending) by their frequency.
2. The highest frequency LHS $c \in C$ is completed as a rule $r : c \to u$ (u is the corresponding RHS which is directly interpreted by manually checking the specifications consisting of c).
3. $R = R \cup \{r\}$; $C = C \setminus \{c\}$.

4. Rules in R then rewrite remaining LHSes in C. When a substitution to $\square$ in $c_i \in C$ occurs, the LHS of c_i is updated.
5. Continue until $C = \varnothing$, a set of rules R is completely obtained.

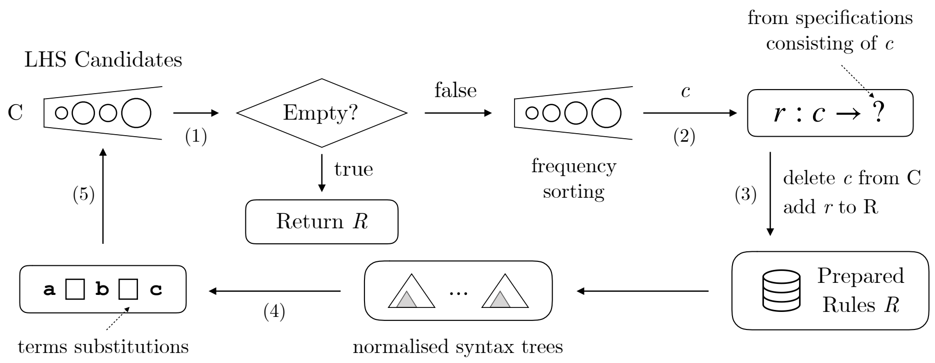

Fig. 4. Recursive rewriting rules preparation

In our experiment, 228 LHSes are automatically extracted. Thereafter, 228 rewriting rules are manually prepared, which is constituted of 208 NP-Phrase rules and 20 Context-Based rules. The number of rules containing conditional LHSes is 85.

4.4 A Comprehensive Example

Recall the sentence S in the section 3. After sequentially normalising the syntax tree, a set of NP-Phrases is obtained $C = \{c_1, c_2, c_3\}$ where c_1 : *first and second operand*, c_2 : *two unsigned 32-bit integer*, and c_3 : *multiply* $\square_2$ *in* $\square_1$. Note that c_3 is obtained by substituting c_2 and c_1 in "*multiply two unsigned 32-bit integer in first and second operand*" by $\square_2$ and $\square_1$, respectively. Since in the syntax tree, lower-degree nodes are practically more likely to occur than higher ones, the frequency ordering is: $c_1 > c_2 > c_3$. Three rules are then prepared as follows:

1. Select c_1 and manually prepare r_1 : *first and second operand* $\rightarrow$ *rn, rm*. By r_1, c_2 is kept unchanged and c_3 is rewritten to $c_3' = \textit{multiply}\ \square_2\ \textit{in rn,rm.}$
2. Select c_2 and manually prepare r_2 : *two unsigned 32-bit integer* $\rightarrow val(\square_3), val(\square_4)$. By r_2, c_3' is rewritten to c_3'' : *multiply* $val(\square_3), val(\square_4)$ *in rn, rm.*
3. Select c_3 and manually prepare r_3 : *multiply* $(val\square_3), val(\square_4)$ *in rn, rm* $\rightarrow mul(val(rn), val(rm))$. Expected rules r_1, r_2, r_3 are now completely obtained.

Note that, if some rules already exist, the preparation simply reuses them. Eventually, when all the rules are prepared, the formal semantics of S represented by a Java statement is interpreted in a bottom-up manner as illustrated in Fig. 5:

$$\begin{aligned} & \text{multiply two unsigned 32-bit integer in } \underline{\text{first and second operand}} \\ \xrightarrow{r_1}\ & \text{multiply } \underline{\text{two unsigned 32-bit integer}} \text{ in rn, rm} \\ \xrightarrow{r_2}\ & \underline{\text{multiply val}(\square_3)\text{, val}(\square_4)\text{ in rn, rm}} \\ \xrightarrow{r_3}\ & \text{mul(val(rn), val(rm))} \end{aligned}$$

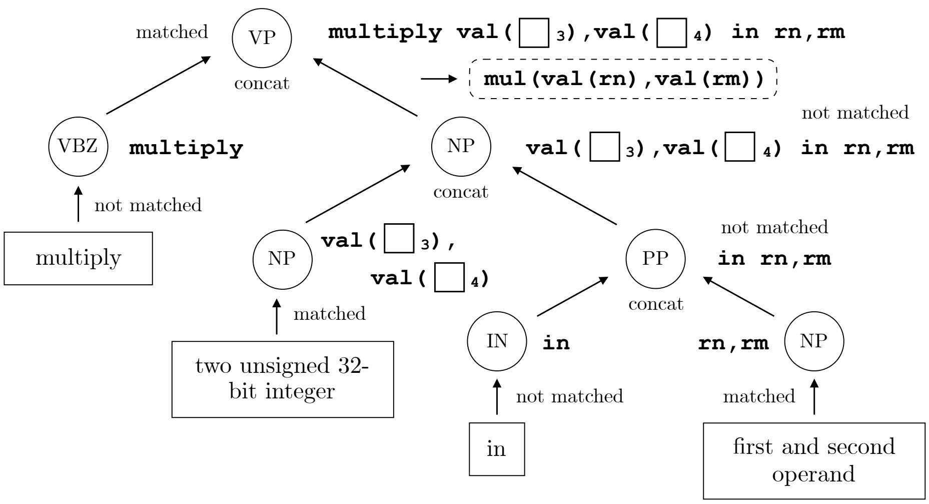

Fig. 5. Semantics interpretation in a bottom-up manner

5 Detecting Modified Flags

Detecting the modification of flags in instructions is practically not straightforward since (1) their descriptions are written totally in natural language and (2) synonyms are diversely used in the *flags-update* sections as indicated in Table 2.

Table 2. The diversity of *flags-update* descriptions

Flags-Update Descriptions	Implications
This instruction does not change the flags	Flags are unchanged
This instruction does not affect the condition code flags	
The V flag is left unmodified.	
This instruction updates the N, Z, C and V flags according to the result.	Modify specific flags
Updates the N and Z flags according to the result. Does not affect the C and V flags.	

Fig. 6 briefly illustrates our proposed solution. Instead of employing a rule-based approach, we adopt a sentences similarity analysis by utilising a well-known topic modeling method called Latent Dirichlet Allocation (LDA) [28]. To train an LDA model, each sentence is firstly represented as a frequency vector of words. Thereafter, when all parameters of the model have already been trained, a *topic* is considered as a distribution of words and a *sentence* is represented as a distribution of topics, which gives their classification based on a similarity measure. Note that, before training the model, each sentence from the *flags-update* section is sequentially normalised by *lemmatisation* and *words refinement* (previously mentioned in section 3). After training (unsupervised) the model by all sentences (1), the topic distribution of a targeted sentence s and the model sentence m = *"update affect set change modify"* are estimated as two dimensional real-number vectors $\overrightarrow{v_s}$, $\overrightarrow{v_m}$, respectively (2). In fact, m is reasonably chosen since it caries a strong meaning of *modified.* The similarity between s and m is then evaluated by calculating the Cosine similarity between $\overrightarrow{v_s}$ and $\overrightarrow{v_m}$ (3). If the result does not exceed a threshold t, s is considered as *modified*, otherwise *unmodified.*

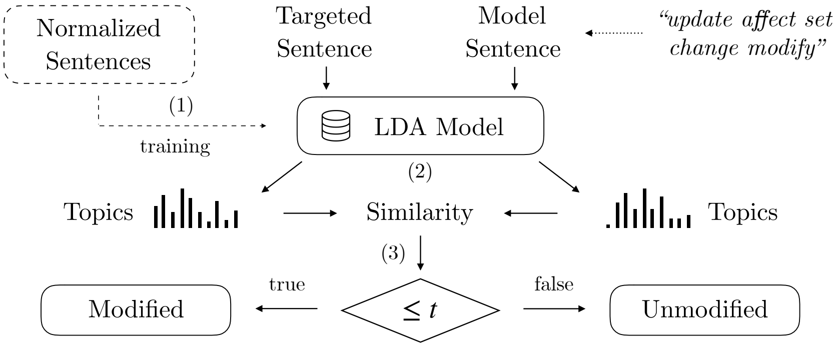

Fig. 6. Detecting modified flags by applying a sentences similarity analysis

Our module utilises an LDA implementation provided in Sklearn [29], in which the hyperparameters are set: $\alpha = 0.1, \beta = 0.1, ntopics = 10, twords = 10, niters = 2000$, and $t = 0.85$. The major advantage behind this approach is that, when extending our method to other architectures, we solely need to redefine t and a new model sentence m, then the algorithm handles the rest. Comparing with rule-based approaches in case applied, this method is obviously more generalised.

6 Conformance Testing

To verify the correctness of a generated Java specification m, we first apply JDART [30] – a dynamic symbolic execution engine built on top of Java Pathfinder [31], to generate a set of test inputs T which covers all feasible execution paths of m. The conformance testing is then performed by *comparing* the execution results of T by m and μVision [20] – a trusted binary emulator supporting numerous ARM variations. Fig. 7 illustrates how our conformance testing works:

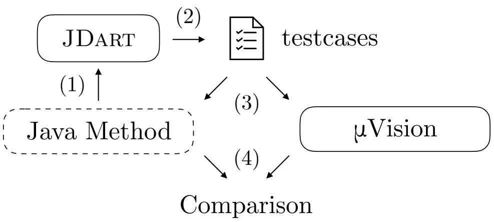

Fig. 7. Conformance testing on the generated Java methods

where: (1) applying dynamic symbolic execution on m, (2) all possible test cases T are generated by JDART, (3) each test case in T is simultaneously executed by m and μVision, and (4) two environments after execution are compared. Finally, if all the test results are passed, it is argued that the correctness of m is verified.

7 The CORANA Tool

The extraction of the formal semantics explicitly implies the generation of a dynamic symbolic execution tool for ARM Cortex-M. By utilising the extracted Java methods, a preliminary version of this tool called CORANA [15] has been developed, which is able to directly interpret and trace obfuscated IoT malware. CORANA takes the advantages of existing powerful engines: CAPSTONE [10] as the single-step disassembler, and Z3 Solver [32] as the back-end theorem prover.

7.1 CORANA Architecture

Fig. 8 depicts a high-level architecture of CORANA, as well as describes how it precisely traces and incrementally reconstructs the Control Flow Graph (CFG) of obfuscated ARM binaries. CORANA is constituted of two main components: (I) An execution kernel provides the semantics framework and the path condition generation, and (II) A symbolic executor, which consists of the generated Java methods built on top of primitive functions, dynamically executes inputted instructions and generates the CFG based on the (in)feasibility of tracing results.

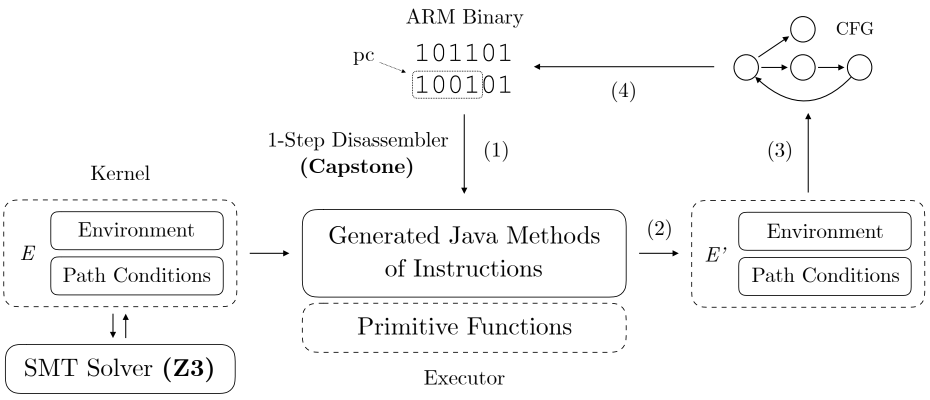

Fig. 8. CORANA architecture

(1) Single-step disassembles the ARM binary file starting from the program counter register pc to obtain an instruction i.
(2) Symbolically executes i, updates the environment and path conditions.
(3) The feasible paths are traced and tested in the depth-first manner, which incrementally generates the CFG.
(4) Repeats disassembling until reaching the end of the binary file or obtaining an unsupported instruction.

To be more specific, step (3) has two main objectives: exploring the destination of indirect jumps, and decrypting self-modifying codes (if exist). Although current IoT malware infrequently contains self-modification, indirect jumps widely occur.

7.2 Path Conditions Generation

By introducing the customised class `BitVec`, the environment transformations is implicitly *embedded* inside the `BitVec` operators. Since `BitVec` computations are totally declared within the primitive functions, the environment is updated without paying an extra effort. As a result, the path conditions are also generated. For instance, the instruction $i : \mathit{subs}\ r_1, r_0, r_1$ sets r_1 by $r_0 - r_1$ and updates flags based on the subtraction. Let r_1, r_0, z are `BitVec` values: $r_1 = \langle x; a\rangle$, $r_0 = \langle y; b\rangle$, $z = \langle z_0; c\rangle$, where x, y, z_0 are `BitSet` values and a, b, c are symbolic values. The semantics transitions on the register r_0, r_1 and the flag z are produced as follows:

$$\begin{aligned}
r_0 &: \langle y; b\rangle \xrightarrow{i} \langle y; b\rangle \\
r_1 &: \langle x; a\rangle \xrightarrow{i} \langle y - x\ ; (\mathit{bvsub}\ b\ a)\rangle \\
z &: \langle z_0; c\rangle \xrightarrow{i} \langle (y - x) == 0\ ; (= (\mathit{bvsub}\ b\ a)\ 0)\rangle
\end{aligned}$$

where *bvsub* is the subtraction operator supported in BitVector theory by default. When a conditional branch occurs at a conditional jump (e.g., *bne*) or a conditional data instruction (e.g., *addne*), Corana symbolically executes this instruction and generates the new path conditions by taking the conjunctions of the pre-condition and the new suffixes at both *true* and *false* branches. For instance, if an instruction *bne* occurs right after the $i : \mathit{subs}\ r_1, r_0, r_1$, Corana adds the suffix *ne* and its negation $\neg ne$ to the current path condition ψ (pre-condition) of *true* and *false* branches, respectively: $(\psi\_true = \psi \wedge (not\ (= (\mathit{bvsub}\ b\ a)\ 0)))$ and $(\psi\_false = \psi \wedge (= (\mathit{bvsub}\ b\ a)\ 0))$ to obtain the post-conditions (in ARM, the conditional suffix *ne* means checking $\neg z$). While executing another instruction i' that constitutes of more complex semantics (e.g., UMALL), the environment transforming and path condition generating become seemingly complicated, but since i' is a *combination* of primitive functions, it will be automatically handled.

8 Experiments

8.1 Semantics Extraction

Table 3 shows the experimental result of the semantics formalisation. Among 1039 collected ARM instructions over 6 variations, the Java specifications of 692 instructions (66.60%) are generated by using only 228 rewriting rules (approximately 0.33 rules are needed to cover an instruction) and 662 of them (63.72%) have passed the conformance testing. We observed two reasons causing failures:

Incorrect modified flags detection. The presence of relatively complex synonym phrases such as *"left unmodified"* confuses our sentences similarity analysis. For instance, the descriptions of *flags-update* sections in the instruction RORS (Cortex-M0 and Cortex-M0+) contain a sentence *"The V flag is left unmodified"*, which is challenging to be correctly distinguished by our method at the moment.

Inappropriate sentences ordering in the operation sections. Our method interprets in a sentence-wise manner, which follows the ordering of sentences in the *operation* section. Thus, if the sentences have an inappropriate order, failures occur. For instance, the instruction STRB (Cortex-M7) is described as *"STRB instruction store a register value into memory. Unsigned byte, zero extend to 32 bits on loads"*, but the correct semantics should be defined in the opposite order: *"STRB instruction zero-extend an unsigned byte value then store into memory"*.

Table 3. The number of successfully extracted semantics over six variations

Variation	Collected	Selected	Generated	Verified
Cortex-M0	63	44 (69.84%)	44 (69.84%)	41 (65.08%)
Cortex-M0+	63	44 (69.84%)	44 (69.84%)	41 (65.08%)
Cortex-M3	129	80 (62.02%)	80 (62.02%)	74 (57.36%)
Cortex-M4	244	167 (68.44%)	167 (68.44%)	161 (65.98%)
Cortex-M7	261	178 (68.20%)	178 (68.20%)	172 (65.90%)
Cortex-M33	279	179 (64.16%)	179 (64.16%)	173 (62.00%)
Total	1039	692 (66.60%)	692 (66.60%)	662 (63.72%)

8.2 Dynamically Handling Jumps by Corana

Since IoT malware rarely contains self-modifications, typical disassemblers (e.g., Capstone and Ida) are able to correctly disassemble them. Nevertheless, when control structures matter, such as VM-aware malware [33] and trigger-based behaviour [34,35], revealing the hidden destination of jumps becomes immensely essential. We describe how Corana traces obfuscated IoT malware by sampling `37c81e` – a `Linux.Mirai` detected by VirusTotal [36], taken from VirusShare [37].

Conditional Jumps. Fig 9 illustrates the presence of a conditional jump `beq` at `0x37648`, where Corana adds `eq` and ¬`eq` to the path conditions of *true* and *false* branches, respectively. Afterwards, Corana detects that these paths are both *feasible* by checking the satisfiability of their symbolic constraints. As a result, instead of solely executing the next instruction at `0x3764C`, Corana additionally traces the *true* branch at `0x37658`, which presents a correct execution behaviour.

Dead Conditional Jumps. Fig. 10 depicts an example of a conditional jump `bne` at `0x5C354`, where Corana detects that the path constraints of the *true* and the *false* branches are *unsatisfiable* and *satisfiable*, respectively. In other words, the *true* branch will be never executed and hence, this jump will be eventually ignored. This is regarded as the opaque predicates obfuscation in IoT malware.

Indirect Jumps. Fig 11 and Fig 12 describe how Corana dynamically handles indirect jumps. At `0x00058`, when a conditional indirect jump `bxeq lr` occurs, Corana adds `eq` and ¬`eq` to the path conditions of *true* and *false* branches, respectively. It then checks the feasibility of these branches and detects that both of them are *feasible.* Especially, by testing with a satisfiable instance at `0x00058`, Corana identifies a possible hidden destination stored in `lr`: `0x0004`.

```
...
0x37640 cmp r5,r1
0x37644 str r2,[r3]
0x37648 beq #0x37658
0x37658 ldr r3,[pc,#0x138]
0x3765C ldr r0,[sp,#8]
...
```

Fig. 9. Conditional jump handling

```
...
0x5C34C cmp r4,r0
0x5C350 moveq r2,r5
0x5C354 bne #0x5c330
0x5C358 b #0x5c334
0x5C35C andeq lr,ip,ip,ror r0
...
```

Fig. 10. Dead jump detection

```
...
0x00050 ldr r1,[r3,r2]
0x00054 cmp r1,#0
0x00058 bxeq lr
0x0005C b #0x60
0x00060 muleq fp,r4,pc
...
```

Fig. 11. Disassembled indirect jump

```
...
0x00050 ldr r1,[r3,r2]
0x00054 cmp r1,#0
0x00058 bxeq lr
0x00004 bl #0x44
0x00044 ldr r3,[pc,#0x14]
...
```

Fig. 12. Indirect jump traced by CORANA

9 Conclusion

Through our study, the feasibility of extracting the formal semantics from natural language specifications has been investigated. To demonstrate this possibility, we present an approach to systematically formalise the semantics of ARM Cortex-M instructions from their official specifications over six variations. Note that, instead of aiming to provide a fully automatic method, our ultimate goal is effectively reducing a large amount of tedious human effort on the implementation of tools relying on formal methods. Additionally, by instantiating the extracted semantics into a prepared framework, a dynamic symbolic execution tool for Cortex-M called CORANA has been preliminarily developed, which is able to correctly trace IoT malware under the presence of obfuscation techniques such as indirect jumps and opaque predicates. We expect our method can be practically extended to other architectures in the same manner without adding complicated modifications. Furthermore, we do hope our work enlightens the ability to leverage the benefits of adopting natural language processing and machine learning to automate rather simple but tedious tasks in the development of formal methods.

Future Directions. Beyond six previously mentioned variations, the proposed method is being considered to apply on other architectures such as MIPS and other ARM Cortex series. Contrary to Cortex-M, the specifications of Cortex-A and Cortex-R are not structurally documented (only PDF files are available on ARM Developer Website at the moment). After parsing the structured data from these PDFs, our approach can be feasibly applied for them in the same manner.

Acknowledgments. We are grateful to Nao Hirokawa, Le Minh Nguyen, and the anonymous reviewers of FM'19 for their insightful feedback and invaluable comments. We sincerely thank Xuan Tung Vu, Thi Hai Yen Vuong, and Lam Hoang Yen Nguyen for their constructive discussions, as well as Thu Trang Hoang for her sharp comments on some grammatical issues. This study is partially supported by JSPS KAKENHI Grant-in-Aid for Scientific Research (B) 19H04083.

References

1. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), pp. 385–394 (1976)
2. Thakur, A., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T. and Reps, T.: Directed Proof Generation for Machine Code. In: Tayssir T., Byron C., Paul J. (eds.) CAV 2010, LNCS, vol. 6174, pp. 288–305. Springer, Heidelberg (2010)
3. Desclaux, F.: Miasm: Framework de reverse engineering. In: Actes du SSTIC (2012)
4. Cha, S.K., Avgerinos, T., Rebert, A. and Brumley, D.: Unleashing Mayhem on binary code. In: IEEE S&P 2012, pp. 380–394 (2012)
5. Anthony, R.: Methods for Binary Symbolic Execution. In: Ph.D Dissertation, Stanford University (December 2014)
6. Bonfante, G., Fernandez, J., Marion, J.Y., Rouxel, B., Sabatier, F. and Thierry, A.,: Codisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions. In: CCS 2015, pp. 745–756 (2015)
7. Hai, N.M., Ogawa, M. and Tho, Q.T.: Obfuscation Code Localization Based on CFG Generation of Malware. In: Joaquin G.A, Evangelos K., Guillaume B. (eds.) FPS 2015, LNCS, vol. 9482, pp. 229–247. Springer, Heidelberg (2015)
8. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C. and Vigna, G.: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: IEEE S&P 2016, pp. 138–157 (2016)
9. Nethercote, N. and Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: ACM PLDI 2007, pp. 89–100 (2007)
10. Capstone Engine, http://capstone-engine.org. Last accessed on 9 July 2019
11. Ida, https://hex-rays.com/products/ida. Last accessed on 9 July 2019
12. Krishnamoorthy, N., Debray, S. and Fligg, K.: Static detection of disassembly errors. In: IEEE WCRE 2009, pp. 259–268 (2009)
13. Dasgupta, S., Park, D., Kasampalis, T., Adve, V.S. and Rosu, G.: A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture. In: ACM PLDI 2019, pp. 1133–1148 (2019)
14. ARM Developer, https://developer.arm.com. Last accessed on 9 July 2019
15. The Corana Tool, https://anhvvcs.github.io/corana. Last accessed on 9 July 2019
16. Robeer, M., Lucassen, G., van der Werf, J.M.E., Dalpiaz, F. and Brinkkemper, S.: Automated Extraction of Conceptual Models from User Stories via NLP. In: IEEE RE 2016, pp. 196–205 (2016)
17. Yue, T., Briand, L.C. and Labiche, Y.: aToucan: An Automated Framework to Derive UML Analysis Models from Use Case Models. ACM TOSEM 24(3), pp. 13:1–13:52 (2015)
18. Heule, S., Schkufza, E., Sharma, R. and Aiken, A.: Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. In: ACM PLDI 2016, pp. 237–250 (2016)
19. Schkufza, E., Sharma, R. and Aiken, A.: Stochastic superoptimization. In: ASPLOS 2013, pp. 305–316 (2013)
20. μVision, http://keil.com/mdk5/uvision. Last accessed on 9 July 2019
21. Yen, N.L.H.: Automatic Extraction of x86 Formal Semantics from Its Natural Language Description. In: Master's Thesis, School of Information Science, JAIST (March 2018)

22. Anh, V.V.: Formal Semantics Extraction from Natural Language Specifications for ARM. In: Master's Thesis, School of Information Science, JAIST (December 2018)
23. Bonfante, G., Marion, J.Y. and Reynaud-Plantey, D.: A Computability Perspective on Self-Modifying Programs. In: SEFM 2009, pp. 231–239 (2009)
24. Degenbaev, U.: Formal Specification of the x86 Instruction Set Architecture. In: Ph.D Dissertation, Universitat des Saarlandes (February 2012)
25. Aceto, L., Fokkink, W. and Verhoef, C.: Structural operational semantics. Handbook of process algebra, pp. 197–292 (2001)
26. Loper, E. and Bird, S.: NLTK: the natural language toolkit. In: ACL (2004)
27. Robertson, S.: Understanding inverse document frequency: on theoretical arguments for IDF. Journal of documentation 60(5), pp. 503–520 (2004)
28. Blei, D.M., Ng, A.Y. and Jordan, M.I.: Latent Dirichlet Allocation. Journal of Machine Learning Research 3, pp. 993–1022 (2003)
29. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V. and Vanderplas, J.: Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research 12, pp. 2825–2830 (2011)
30. Luckow, K., Dimjasevic, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamaric, Z. and Raman, V: JDart: A Dynamic Symbolic Analysis Framework. In: Marsha C., Jean-Francois R. (eds.) TACAS 2016, LNCS, vol. 9636, pp. 442–459. Springer, Heidelberg (2016)
31. Visser, W., Havelund, K., Brat, G., Park, S. and Lerda, F.: Model checking programs. Automated software engineering 10(2), pp. 203–232 (2003)
32. De Moura, L. and Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan C.R., Jakob R. (eds.) TACASE 2008, LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008).
33. Kirat, D., Vigna, G. and Kruegel, C.: BareBox: Efficient malware analysis on bare-metal. In: ACSAC 2011, pp. 403–412 (2011)
34. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D. and Yin, H.: Automatically identifying trigger-based behavior in malware. In: Wenke L., Cliff W., David D. (eds.) Botnet Detection 2008, ADIS, vol. 36, pp. 65–88. Springer, Heidelberg (2008).
35. Fleck, D., Tokhtabayev, A., Alarif, A., Stavrou, A. and Nykodym, T.: PyTrigger: A System to Trigger & Extract User-Activated Malware Behavior. In: AERES 2013, pp. 92–101 (2013)
36. Virus Total, https://www.virustotal.com. Last accessed on 9 July 2019
37. Virus Share, https://virusshare.com. Last accessed on 9 July 2019