Towards A Binary Intermediate Language for Real-Time Embedded System

Jianqi Shi*, Qin Li†, Longfei Zhu‡, Xin Ye†, Yanhong Huang*, Huixing Fang† and Fu Song†
*National Trusted Embedded Software Engineering Technology Research Center, East China Normal University, China
‡School of Computer Science and Technology, Donghua University, China,
†Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China
Email: {*jqshi,†qli,†xye,*yhhuang,†wxfang,†fsong}@sei.ecnu.edu.cn, ‡lfzhu@dhu.edu.cn

***Abstract*—In this paper, we present a novel binary intermediate language - `xBIL` for supporting the analysis and verification of embedded systems. Time and interrupt behavior features are taken into account in this language. Meanwhile, the syntax, operational and denotational semantics, bit-vector arithmetic operations and the bit-vector formula decision procedure are presented for this intermediate language. The language has been applied to several practical cases. One significant application of this language is assisting the analysis and verification of practical commercial, automotive operating system. Through the practical application, the language proposed in this paper shows its expressiveness and the flexibility.**

***Index Terms*—Binary Intermediate Language, Semantics, Bit-Vector Arithmetic, Decision Procedure**

I. Introduction

With the rapid development of software engineering, the correctness and robustness of software become more and more important, especially in some critical applications. As a result, plenty of modelling approaches and tools are investigated to build formal models which can be applied to analysis and improvement of the software quality, especially, for real-time embedded system. For analysing and verifying the software module, usually, researchers can solve the relevant problems from source code or binary code perspectives. Most of the research works focus on source code level. One reason is the uniformity of source code simplifies the analysis work and brings unambiguous semantics to the developers. Nonetheless, in some special scenarios, the binary code level analysis is necessary and important. Compared with the source code, binary code corresponds with the instructions that are running on the processors. The errors caused by compilers and optimisation are reserved in the binary code. Thus, it is very significant to detect the potential vulnerabilities in binary code level. Meanwhile, users such as the components integrators, some system testers and end users only have the binary executables or libraries of the software. If they want to evaluate the software components or check whether the libraries run correctly, perhaps the binary code level analysis and verification are the most straightforward ways. However, in binary code level, the codes are hardware platform dependent. Without intermediate representation, it is very difficult to analyse and verify the software module within a unified technical framework. Thus, the intermediate representation of binary code is of vital importance for all the researchers and developers focusing on binary code level.

Qin Li is the corresponding author with the Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, 3663 Zhongshan Road North, Shanghai, China.

Binary code analysis has been studied a lot recently. There is a variety of platforms such as CodeSurfer/x86 [1], McVeto [2], Phoenix [3], and Jakstab [4] which firstly disassembled binary code into assembly instructions, lifted the instructions to an intermediate language (IL), and then performed analysis in the IL level. Within BitBlaze project [5], a unified binary analysis platform was proposed to ensure the security of the applications. BitBlaze focuses on the root cause of the problems and extracts security-related properties from binary programs directly. CodeSurfer/x86 [1] provides a way to recover intermediate representations and information about the contents of memory locations, which can be used for analysing the x86 executable and investigating the behaviours of potential malicious code. Caballero et al. [6] focused on the automated binary code reuse and have developed an approach to identifying the prototype of an undocumented code fragment directly from the binary code. Lee et al. [7] presented an accurate and precise way to reconstruct high-level language data type abstractions from binary code. These facts have shown that it is practical and powerful to launch the verification in the domain of binary code. Lundqvist et al. [8] implemented a CPU simulator to perform the analysis related to pipeline, cache and other hardware components. It offers a good way to simulate the execution of processors, but it is not portable due to the sophisticated model of the CPU and the restriction in the analysis. It is noteworthy that a project called LLVM [9] (Low Level Virtual Machine) provides a compiler framework designed to support transparent, lifelong program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs. As of now, the LLVM is becoming the most popular compilation infrastructure for modern programming languages.

So far, most of the BILs focus on Malware Analysis, Software Verification and Test Case Generation. However, hardware behaviour cannot be described in binary code exactly so that it brings more obstacles to related verification work. For example, interrupt behaviours have not been supported in other BILs yet. However, to ensure the interrupt safety is vital in the process of verifying software system whose safety property is quite important. Moreover, verification on the binary code of cross-platform software system requires different formal specifications based on different hardware platforms. Meanwhile, for simulating the system behaviours and reasoning about the functional or non-functional properties, it is better to define the accurate operational and denotational semantics of the intermediate language. However, only a small number of the existing binary intermediate languages have the operational semantics, let alone define the denotational semantics. Some compilation infrastructures provide intermediate languages to unify the expression of the processing programming languages. In this case, the semantics of this language cannot be specified comprehensively. For example, for the LLVM code, the embedded assembly code in source code cannot be processed. LLVM does not know the exact semantics of these assembly codes.

In order to tackle with the problems mentioned above, we proposed a new binary intermediate language - `xBIL` for analysing and verifying the safety and time sensitive properties. The contributions of our work can be summarised as follows: 1) We propose a novel binary intermediate language for expressing the binary code on multiple hardware platforms. This new language supports interrupt and time relevant features. Additionally, this intermediate language can declare the runtime environment. This feature helps users to expand the usage scenarios and the supported processors in the future. 2) We provide accurate operational and denotational semantics of the language we proposed. Operational semantics depicts the execution mechanism, and can be applied to simulate a specific `xBIL` program. The denotational semantics of `xBIL` program uses mathematical objects to define the behaviours of the program. 3) All the expressions in `xBIL` programs are designed as bit-vector oriented, for making our framework more flexible. This design brings new challenges for reasoning about the program properties. A novel framework of expressing the bit-vector arithmetic operations and deciding the bit-vector relevant logic formulas is desired. We present a unified approach for presenting the bit-vector arithmetic operations. Afterwards, we develop the existing bit-vector decision procedures and propose a more efficient algorithm for solving the `xBIL` bit-vector formulas. 4) We have applied `xBIL` on the binary code verification of practical commercial, automotive operating system - ORIENTAIS [10]. We have successfully found dozens of bugs even after strict testing. This application helps to prove the feasibility and efficiency of our approach.

This work is extended from our previous paper [11]. In this paper, we simplify the syntax of `xBIL` and define the denotational semantics. Additionally, the bit-vector arithmetic rules and the decision procedure of the bit-vector formula are proposed in this paper as well.

The remainder part of this paper is organised as follows: We introduce the syntax of `xBIL` in Section II. Next, we give the operational and denotational semantics of `xBIL` in Section III. Bit-vector relevant information including arithmetic operations and bit-vector formula decision procedure are discussed in Section IV. We report the details regarding the application in Section V. In the end, we summarise this paper in Section VI.

II. The Syntax of xBIL

In this section, we will introduce the syntax of `xBIL`. The details are divided into two parts: the runtime environment declaration and the intermediate language statements. The runtime environment declaration depicts the hardware platform for running the `xBIL` program. Usually, these information include the specific features of target microprocessor. For instance, the machine word length, entry point address, register names, interrupts and other hardware relevant information should be listed by using the runtime environment declaration. The intermediate language statements which are composed by multiple `xBIL` commands act as the common imperative programs. In the following part, the syntax of runtime environment declaration and intermediate language statements will be listed respectively.

A. *Runtime Environment Declaration*

In the syntax of runtime environment declaration, entry address, machine word, registers, memory blocks, variables, labels and interrupt service routines are defined. The syntax details are given as follows:

$$\begin{aligned}
\texttt{RTEnv}_{decl} ::= {} & \texttt{entry addr;} \\
& \texttt{memory[addr}_1 \rightarrow \texttt{val}_1\texttt{:}(\tau_{reg1},\texttt{e}_{t1}), \\
& \qquad \cdots, \\
& \qquad \texttt{addr}_n \rightarrow \texttt{val}_n\texttt{:}(\tau_{regn},\texttt{e}_{tn})]; \\
& \texttt{machineword}\ \tau_{reg}; \\
& (\texttt{register reg}_{id}\texttt{:}\tau_{reg};)^* \\
& (\texttt{var var}_{id}\texttt{@addr:}(\tau_{reg}, e_t););)^* \\
& (\texttt{mem mem}_{id}\texttt{@addr,val:}(\tau_{reg}, e_t);)^* \\
& (\texttt{label label}_{id}\texttt{@addr;})^* \\
& (\texttt{irq irq}_{number} \rightarrow (\texttt{label} \mid \texttt{addr});)^*
\end{aligned}$$

$$\begin{aligned}
\tau_{reg} ::= {} & reg1_t : 1 \mid reg8_t : 8 \mid reg16_t : 16 \mid \\
& reg32_t : 32 \mid reg64_t : 64 \mid \tau_{reg} + \tau_{reg}
\end{aligned}$$

$$\texttt{reg}_{id} ::= \texttt{string}$$

$$\texttt{addr} ::= \texttt{integer}$$

$$e_t ::= \texttt{little:0} \mid \texttt{big:1}$$

τ_{reg} represents the register value type and can be a variety of value types such as 1 bit, 8 bits, $\cdots$, 64bits as it is shown in the syntax definition. Meanwhile, the operator '+' is used to extend the value type. Taking $reg8_t + reg8_t$ as an example, it represents a $reg16_t$ type. The extended register value type can be used on the special occasion. $\texttt{reg}_{id}$ is the register name defined by a string. `addr` ::= `integer` presents that every xBIL address is integer. e_t indicates the endian type of one memory value. Usually, only `little` and `big` can be chosen as the endian format type.

`entry addr` defines the entry point of the program, where the `addr` is the concrete address of the entry point. The xBIL instructions which implement the equivalent functions of one specific machine instruction will be related to the corresponding address of this machine instruction `Entry addr` pointed to the xBIL instruction address corresponds to the first machine instruction of binary code. `memory` [$\texttt{addr}_1 \rightarrow \texttt{val}_1$:($\tau_{reg1}$,$\texttt{e}_{t1}$), ... , $\texttt{addr}_n \rightarrow \texttt{val}_n$:($\tau_{regn}$,$\texttt{e}_{tn}$)] predefines the values of memory in the initial state. The related value for each address `addr` is predefined as `val` in the global system memory before the `xBIL` program starts. `val`:(τ_{reg},$\texttt{e}_t$) defines the type of memory value located at `addr` as τ_{reg}. Meanwhile, the corresponding endian type of this memory value is e_t. `machineword` τ_{reg} defines the machine word of target processor where the binary program runs. `register` $\texttt{reg}_{id}$:τ_{reg} declares a register whose value type is τ_{reg}. `var` $\texttt{var}_{id}$@`addr`:(τ_{reg}, e_t) defines a variable named $\texttt{var}_{id}$ located at address `addr` with type τ_{reg}. The purpose of using variables in the intermediate language is to reuse the decompiled resources which have resolved some variables by parsing the exported symbol table. Similarly, the way endian type is defined by e_t. `mem` $\texttt{mem}_{id}$@`addr`,`val`:(τ_{reg}, e_t) denotes a memory block variable named as $\texttt{mem}_{id}$ and starts from `addr`. τ_{reg} specifies value type of all the elements and the element value assignment operations follow the endian type e_t. While lifting the `xBIL` program from binary code, for convenience, we usually name some instruction locations with read-friendly aliases. `label` $\texttt{label}_{id}$@`addr` makes it possible. We use `irq` $\texttt{irq}_{number}\rightarrow$(`label` | `addr`) to define a potential interrupt and its corresponding interrupt service routine entry address. After discussing the runtime environment declaration, the intermediate language statements and the corresponding commands will be introduced in the following part.

B. The Syntax of xBIL Statements

The `xBIL` statements are composed by three parts. They are programs definition, expression definition and instruction definition. Program definition defines the sequence of `xBIL` instructions. Expression definition shows the usage of composing an `xBIL` expression. All the relevant elements in the `xBIL` expressions are built upon the bit-vector arithmetic operations. The detailed instructions will be presented in Section IV. The instruction definition will explain the execution mechanism of `xBIL`. Here gives the syntax of all the three parts as follows:

1) Program Syntax: The `xBIL` program is a sequence of instructions. For each instruction, the address and command are the components to identify the location and function of this instruction. As the core concept of `xBIL` is simulating one practical hardware instruction by using a sequence of `xBIL` commands, thus not all the commands will be related to the location of original hardware instruction. For this reason, the address can be either integer or $\perp$, where $\perp$ denotes an empty address. Only the first instruction of the instruction sequence corresponded to one hardware instruction will have the actual address value. The formal definition is given in the following list:

$$\begin{aligned} \texttt{program} &::= (\texttt{instruction})^+ \\ \texttt{instruction} &::= \texttt{address:command}; \\ \texttt{address} &::= \textit{integer} \;\mid\; \perp \end{aligned}$$

2) Expression Syntax: For constructing the expressions of `xBIL` programs, we employed multiple operators. All these operators are bit-vector oriented, which means these operators can only process the bit-vector inputs and output the corresponding calculation result in the form of bit-vector. Here lists the detailed syntax of the expressions:

$$\texttt{exp} ::= \texttt{exp} \diamond_{bvop} \texttt{exp} \mid \texttt{number}_{bv} \mid \texttt{reg}_{id} \mid \perp \mid \texttt{var}_{id} \mid$$

$$\texttt{mem}_{id}\texttt{[F}_{decN}\texttt{(exp)]} \mid \texttt{memory(exp,}\tau_{reg}\texttt{,}e_t\texttt{)}$$

$$\diamond_{bvop} ::= +_{[l]} \mid -_{[l]} \mid *_{[l]} \mid /_{[l]} \mid \texttt{mod}_{[l]} \mid \ll \mid \gg \mid !_{[l]} \mid |_{[l]} \mid \ldots$$

`exp` $\diamond_{bvop}$ `exp` is the calculation result of two bit-vector expressions. $\diamond_{bvop}$ is the operator allowed by the engaged bit-vectors. Addition, subtraction, multiplication, division, mod, left/right shift and many other logic operations are supported by `xBIL` expressions. Section IV will explain the detailed function description and the corresponding semantics of all the supported bit-vector arithmetics operators. `memory(exp,`τ_{reg}`,`e_t`)` evaluates the value located at the given address represented by `exp` as a bit-vector, where τ_{reg} is the value type. $\texttt{mem}_{id}\texttt{[F}_{decN}\texttt{(exp)]}$ is the bit-vector on a named memory block with the offset specified by $\texttt{F}_{decN}\texttt{(exp)}$. $\texttt{F}_{decN}$ is a decoding function which maps a bit-vector to its corresponding positive number. In this case, the input integer of function $\texttt{F}_{decN}$ is evaluated by `exp`. It is clear to see that var_{id} and reg_{id} indicate the bit-vector corresponds to the value of variable var_{id} and register reg_{id}. $\perp$ denotes the non-deterministic value in the memory, it is commonly used to identify an un-initialised memory block.

3) Command Syntax: The commands here are not the real sense of instructions running on the CPU. The xBIL provides commands for simulating the behaviours of executing an instruction.

$$\begin{aligned} \texttt{command} ::= \; & \texttt{reg}_{id}\texttt{:=exp} \mid \texttt{var}_{id}\texttt{:=exp} \mid \texttt{exp}\triangleright\texttt{command} \mid \\ & \texttt{halt} \mid \texttt{cost integer} \mid \\ & \texttt{mem}_{id}\texttt{[F}_{decN}\texttt{(exp}_1\texttt{)]:=exp}_2 \mid \\ & \texttt{write(exp}_1\texttt{, exp}_2\texttt{, }\tau_{reg}\texttt{, }e_t\texttt{)} \mid \\ & \texttt{jmp (label}_{id} \mid \texttt{addr} \mid \texttt{exp)} \mid \\ & \texttt{checkirq} \mid \texttt{raise irq} \mid \\ & \texttt{enable}_{irq}\{\texttt{irq}_1\texttt{, irq}_2\texttt{,}\ldots\} \mid \\ & \texttt{disable}_{irq}\{\texttt{irq}_1\texttt{, irq}_2\texttt{,}\ldots\} \end{aligned}$$

`reg`$_{id}$`:=exp` indicates the assignment on the register named `reg`$_{id}$. `var`$_{id}$`:=exp` implements the similar function that set the evaluated value to variable `var`$_{id}$ instead of register. `exp`$\triangleright$`command` is a conditional execution command which invokes the `command` when the evaluated value of `exp` equals to `1`. `halt` terminates the program, then no more instructions can be executed further. `cost integer` indicates the time consumption counter of the hardware instruction. The `integer` is the consumption value, and it marks the time units costed by the hardware instruction which this command implements. Taking `cost 2` for example, it means the denoted instruction consumes two time units for executing the given hardware instruction. `mem`$_{id}$`[F`$_{decN}$`(exp`$_1$`)]:=exp`$_2$ denotes the assignment of one memory block element. The offset of this memory block is specified by `F`$_{decN}$`(exp`$_1$`)`, and the chosen element's value is set to a value expressed by `exp`$_2$. `write(exp`$_1$`, exp`$_2$`,` τ_{reg}`,` e_t`)` writes the evaluated value of exp_2 to the address represented by exp_1, where τ_{reg} and $endian_{type}$ are given to specify the value type and endian format. `jmp (label`$_{id}$ `| addr | exp)` updates the control point of the running program to a new location which is specified by label$_{id}$, `addr` or `exp`. This command evaluates the actual value of the address identifiers, then jumps to target location. `checkirq` checks whether there is an incoming interrupt request. In the case one request exists, the control point will be updated according to address declared in runtime environment declaration. `raise irq` triggers an interrupt event which is specified by `irq` and sends out the corresponding interrupt request. `enable`$_{irq}$$\{$`irq`$_1$`, irq`$_2$`,...`$\}$ enables a series of interrupts. $\{$`irq`$_1$`, irq`$_2$`,...`$\}$ is a set of interrupts which are desired to be enabled. On the contrary, `disable`$_{irq}$$\{$`irq`$_1$`, irq`$_2$`,...`$\}$ disables the given interrupts.

C. Code Example

The `xBIL` code can be used to express variety of instructions for multiple hardware platforms. We have successfully applied `xBIL` to the analysis of `ARM`, `DSP` and `x86` instruction sets. In Table I, we demonstrate the instruction comparison of `x86` assembly code and `xBIL` intermediate code. The registers `ZF`, `SF` and `OF` are the status after the execution of one instruction. They are designed to fetch additional information of the execution. For different hardware platform, the registers are different.

In this section, we presented the syntax of `xBIL` program. Further information regarding the semantics will be discussed in the next section.

III. Operational and Denotational Semantics

As we introduced in previous sections, `xBIL` is an expressive binary intermediate language for implementing the embedded systems. The syntax shows the feasibility of describing multiple hardware platforms. In this section, we will focus on both the operational and denotational semantics of `xBIL`. The operational semantics is the most important base to perform the simulation. The denotational semantics is a key preparation work for verifying the program written in `xBIL` in theorem proving way. We will firstly discuss the operational semantics, afterwards, the denotational semantics will be talked about.

A. Operational Semantics

For operational semantics of the instructions, its transition rules are written in the notation $C \rightsquigarrow C'$ where C and C' are the configurations before and after an execution respectively. We stipulate that if an element in the tuple does not change in one execution, the element can be omitted in the transition rule. Before detailing the semantics, we introduce the symbols used in the semantics definitions first.

$\sigma_{\mathbb{R}}$:$\{reg_{id}\}\rightsquigarrow bv$ is the evaluation function that maps the register name to its value bit-vector. $\sigma_{\mathbb{M}}$:$\{addr\}\rightsquigarrow bv_{[8]}$ maps the address of the memory to the corresponding byte. Now, we define the data state σ of the program as a pair: σ=($\sigma_{\mathbb{R}}$,$\sigma_{\mathbb{M}}$). Σ is the set of all the σ. σ_{halt} is the terminate state. The program terminates when reaches this state. $\sigma_{\mathbb{M}}[addr_1,addr_2,\ldots,addr_n] \rightsquigarrow bv_{[8n]}$ reads the bytes from address $addr_1$ one by one, and combines all these bytes as an $8n$ length bit-vector. $\langle exp, \sigma\rangle \rightsquigarrow bv$ evaluates the bit-vector result of exp under the state σ. Δ:$var_{id}\rightarrow addr$ connects the variable names and the corresponding addresses. $\mathbb{L}$:$label_{id}\rightarrow addr$ gets the address values by the label names.

$\langle command,\sigma\rangle\rightsquigarrow\sigma'$ means the state σ of the program will be updated to σ' after executing the $command$. $\Pi[addr]$ extracts the instruction sequence after address $addr$. $\sigma[\sigma_{\mathbb{R}}[rbv1_1/reg_{id}],\ldots,\sigma_{\mathbb{M}}[mbv/addr_1,\ldots,mbv/addr_n]]$ firstly updates the value of register reg_{id} to $rbv1_1$ and replaces the content of the memory block which is specified by the location sequence $addr_1,\ldots,addr_n$ to mbv, then returns the updated state as a result. `bytes`:$\{\tau_{reg}\} \rightarrow n$ returns the byte length of the given register data type. `mem`$_{id}$$\downarrow_{valtype}\rightsquigarrow vt$ gets the data type of the given memory block, while `mem`$_{id}$$\downarrow_{endian_t}\rightsquigarrow e_t$ returns the endian format. `mem`$_{id}$$\downarrow_{address}\rightsquigarrow addr$ gives the start address of memory `mem`$_{id}$, and `var`$_{id}$$\downarrow_{address}\rightsquigarrow addr$ is a similar function that returns the start address of a variable. var$_{id}$$\downarrow_{address}\rightsquigarrow addr$ and var$_{id}$$\downarrow_{endian_t}\rightsquigarrow e_t$ are very similar to the corresponding operations of memory block, and returns the data type as well as endian format.

$\langle command, T_{total}\rangle\rightsquigarrow T'_{total}$ updates the total time cost value after executing the $command$. IRQ is the current enabled interrupts set. $IRQ\cup\{irq_1,irq_2,...\}$ adds the set $\{irq_1,irq_2,...\}$ to the enabled interrupts set. $IRQ-\{irq_1,irq_2,\ldots\}$ disables the given set of interrupts. $\langle command, IRQ\rangle\rightsquigarrow IRQ'$ assigns a new enabled interrupts set to the running program after executing the given $command$. IVT:$irq \rightarrow addr$ helps to obtain the service routine address of the triggered interrupt irq. $\iota(IRQ)\rightsquigarrow IRQ'_{MASK}$ selects an interrupt from the enabled interrupt set following some user given policies and assigns this selected interrupt to IRQ'_{MASK}. This function is usually implemented by user for simulating the execution

1	012F13DE	mov	dword ptr [sum],0	012F13DE:	sum:=$F_{encN[32]}(0)$;
2	012F13E5	mov	dword ptr [i],1	012F13E5:	i:=$F_{encN[32]}(1)$;
3	012F13EC	jmp	wmain+37h (12F13F7h)	012F13EC:	jmp 012F13F7;
4	012F13EE	mov	eax,dword ptr [i]	012F13EE:	EAX: = i;
5	012F13F1	add	eax,1	012F13F1:	EAX: = EAX $+_{[32]}$ $F_{encN[32]}(1)$;
6	012F13F4	mov	dword ptr [i],eax	012F13F4:	i:= EAX;
7	012F13F7	cmp	dword ptr [i],64h	012F13F7:	. . .
8	012F13FB	jg	wmain+48h (12F1408h)		i $>_{[32]}$ $F_{encN[32]}(100) \triangleright$ ZF:=0;
9	012F13FD	mov	eax,dword ptr [sum]		i $>_{[32]}$ $F_{encN[32]}(100) \triangleright$ SF:=1;
10	012F1400	add	eax,dword ptr [i]		i $>_{[32]}$ $F_{encN[32]}(100) \triangleright$ OF:=1;
11	012F1403	mov	dword ptr [sum],eax		. . .
12	012F14xx	. . .		012F13FB:	(SF $=_{[1]}$ 1) $\wedge_{[1]}$ (ZF$=_{[1]}$ 0) $\wedge_{[1]}$ (OF$=_{[1]}$=1) $\triangleright$ jmp 12F1408h;
13	012F14xx	. . .		012F13FD:	EAX:= sum;
14	012F14xx	. . .		012F1400:	EAX:= EAX $+_{[32]}$ F_{encN}[32](1);
15	012F14xx	. . .		012F1403:	sum:= EAX;

TABLE I
X86 ASSEMBLY CODE AND THE CORRESPONDING XBIL CODE

environment. $\sigma[IRQ'_{MASK}/IRQ_{MASK}]\rightsquigarrow\sigma'$ replaces the current interrupt mask IRQ_{MASK} to IRQ'_{MASK}.

Operational Semantics of Expression: Firstly, we introduce the semantics of `xBIL` expression as follows:

$$\langle \mathbf{e_1} \ \diamond_{\mathbf{bvop}} \ \mathbf{e_2}, \sigma\rangle \rightsquigarrow \mathbf{bv}, \text{ where} \\ \langle e_1, \sigma\rangle \rightsquigarrow bv_1,\ \langle e_2,\ \sigma\rangle \rightsquigarrow bv_2 \ and \ bv = bv_1 \diamond_{bvop} bv_2 \tag{OS-1}$$

$$\langle \mathbf{number_{bv}}, \sigma\rangle \rightsquigarrow \mathbf{number_{bv}} \tag{OS-2}$$

$$\langle \mathbf{reg_{id}}, \sigma\rangle \rightsquigarrow \sigma_{\mathbb{R}}(\mathbf{reg_{id}}) \tag{OS-3}$$

The bit-vector value of register reg_{id} is evaluated to bv.

$$\langle \mathbf{var_{id}}, \sigma\rangle \rightsquigarrow \mathbf{memory(addr, vt, e_t)}, \text{ where} \\ var_{id} \downarrow_{valtype} \rightsquigarrow vt, var_{id} \downarrow_{endian_t} \rightsquigarrow e_t, \Delta(var_{id}) = addr \tag{OS-4}$$

$$\langle \mathbf{mem_{id}[e]}, \sigma\rangle \rightsquigarrow \\ \mathbf{memory(addr + n * F_{decN}(offset), vt, e_t)}, \text{ where} \\ mem_{id} \downarrow_{address} \rightsquigarrow addr, mem_{id} \downarrow_{endian_t} \rightsquigarrow e_t, \\ \langle e, mem_{id} \downarrow_{valtype} \rightsquigarrow vt, \sigma\rangle \rightsquigarrow offset, bytes(vt) = n \tag{OS-5}$$

$$\langle \mathbf{memory(exp, reg_t, e_t)}, \sigma\rangle \rightsquigarrow \mathbf{v}, \text{ where} \\ \langle exp, \sigma\rangle \rightsquigarrow addr, bytes(reg_t) = n,\ e_t = little, \\ \sigma_{\mathbb{M}}[F_{decN}(addr) + n - 1, \ldots, F_{decN}(addr)] \rightsquigarrow v \tag{OS-6}$$

$$\langle \mathbf{memory(exp, reg_t, e_t)}, \sigma\rangle \rightsquigarrow \mathbf{v}, \text{ where} \\ \langle exp, \sigma\rangle \rightsquigarrow addr,\ bytes(reg_t) = n,\ e_t = big, \\ \sigma_{\mathbb{M}}[F_{decN}(addr), \ldots, F_{decN}(addr + n - 1)] \rightsquigarrow v \tag{OS-7}$$

The operational semantics of `xBIL` expression depicts the transition mechanism of the evaluation process. For each operation, firstly, we will give the transition rules. Next, the condition of this transition will be listed. For simplicity, we ignore the rules of bit-vector arithmetic operators. Instead, we list a normal form for the rules of all the operators. In the following part, we will talk about the operational semantics of `xBIL` commands.

$$\langle \mathbf{c_0; c_1}, \sigma\rangle \rightsquigarrow \sigma', \text{ where} \\ \langle c_0, \sigma\rangle \rightsquigarrow \sigma'',\ \langle c_1, \sigma''\rangle \rightsquigarrow \sigma' \tag{OS-8}$$

$$\langle \mathbf{reg_{id} := exp}, \sigma\rangle \rightsquigarrow \sigma[\sigma_{\mathbb{R}}[\mathbf{rbv/reg_{id}}]], \text{ where} \\ \langle exp, \sigma\rangle \rightsquigarrow rbv \tag{OS-9}$$

$$\langle \mathbf{var_{id} := exp}, \sigma\rangle \rightsquigarrow \sigma', \text{ where} \\ var_{id} \downarrow_{endian_t} \rightsquigarrow e_t,\ var_{id} \downarrow_{valtype} \rightsquigarrow vt,\ \Delta(var_{id}) = addr, \\ \langle exp, \sigma\rangle \rightsquigarrow bv,\ \langle write(addr, bv, vt, e_t), \sigma\rangle \rightsquigarrow \sigma' \tag{OS-10}$$

$$\langle \ \mathbf{exp} \ \triangleright \mathbf{command}, \sigma\rangle \rightsquigarrow \sigma, \text{ where} \\ \langle exp, \sigma\rangle \rightsquigarrow bv,\ bv = \{0\} \tag{OS-11}$$

$$\langle \ \mathbf{exp} \ \triangleright \mathbf{command}, \sigma\rangle \rightsquigarrow \sigma', \text{ where} \\ \langle exp, \sigma\rangle \rightsquigarrow bv,\ bv \neq \{0\},\ \langle command, \sigma\rangle \rightsquigarrow \sigma' \tag{OS-12}$$

$$\langle \mathbf{halt}, \sigma\rangle \rightsquigarrow \sigma_{\mathbf{halt}} \tag{OS-13}$$

$$\langle \mathbf{mem_{id}[e_1] := e_2}, \sigma\rangle \rightsquigarrow \sigma', \text{ where} \\ \langle write(addr + offset * n, bv, vt, e_t), \sigma\rangle \rightsquigarrow \sigma', \\ \langle e_1, \sigma\rangle \rightsquigarrow offset,\ bytes(vt) = n, \langle e_2, \sigma\rangle \rightsquigarrow bv, \\ mem_{id} \downarrow_{address} \rightsquigarrow addr,\ mem_{id} \downarrow_{valtype} \rightsquigarrow vt \\ mem_{id} \downarrow_{endian_t} \rightsquigarrow e_t, \tag{OS-14}$$

$$\langle \mathbf{cost\ n}, \sigma\rangle \rightsquigarrow \sigma, \text{ at the same time} \\ \langle \mathbf{cost\ n, T_{total}}\rangle \rightsquigarrow \mathbf{T_{total} + n} \tag{OS-15}$$

$$\langle \mathbf{write(exp_1, exp_2, vt, e_t)}, \sigma\rangle \rightsquigarrow \\ \sigma[\sigma_{\mathbb{M}}[\mathbf{mbv/F_{decN}(addr) + n, \ldots, F_{decN}(addr)}]], \text{ where} \\ \langle exp_1, \sigma\rangle \rightsquigarrow addr, bytes(vt) = n, \\ \langle exp_2, \sigma\rangle \rightsquigarrow mbv, e_t = little \tag{OS-16}$$

$$\langle \mathbf{write(exp_1, exp_2, vt, e_t)}, \sigma\rangle \rightsquigarrow \\ \sigma[\sigma_{\mathbb{M}}[\mathbf{mbv/F_{decN}(addr), \ldots, F_{decN}(addr + n)}]], \text{ where} \\ \langle exp_1, \sigma\rangle \rightsquigarrow addr, bytes(vt) = n, \\ \langle exp_2, \sigma\rangle \rightsquigarrow mbv, e_t = big \tag{OS-17}$$

$$\langle \mathbf{jmp\ label_{id}; commands}, \sigma\rangle \rightsquigarrow \sigma', \text{ where} \\ \langle \Pi[\mathbb{L}(label_{id})], \sigma\rangle \rightsquigarrow \sigma' \tag{OS-18}$$

$$\langle \mathbf{jmp\ addr; commands}, \sigma\rangle \rightsquigarrow \sigma', \text{ where} \\ \langle \Pi[addr], \sigma\rangle \rightsquigarrow \sigma' \tag{OS-19}$$

$$\begin{aligned}&\langle \mathbf{checkirq}, \sigma\rangle \rightsquigarrow \sigma'', \text{ where}\\ &\iota(IRQ) \rightsquigarrow IRQ'_{MASK}\ IRQ'_{MASK} > IRQ_{MASK} \neq 0,\\ &\sigma[IRQ'_{MASK}/IRQ_{MASK}] \rightsquigarrow \sigma',\\ &\langle \Pi[IVT(IRQ_{MASK})], \sigma'\rangle \rightsquigarrow \sigma''\end{aligned} \tag{OS-20}$$

$$\begin{aligned}&\langle \mathbf{checkirq}, \sigma\rangle \rightsquigarrow \sigma, \text{ where}\\ &\iota(IRQ) \rightsquigarrow IRQ'_{MASK},\ 0 \leq IRQ'_{MASK} \leq IRQ_{MASK}\end{aligned} \tag{OS-21}$$

$$\begin{aligned}&\langle \mathbf{raise\ irq}, \sigma\rangle \rightsquigarrow \sigma', \text{ where}\\ &irq > IRQ_{MASK},\ IRQ \cap irq \neq \emptyset,\\ &\langle \Pi[IVT(IRQ_{MASK})], \sigma\rangle \rightsquigarrow \sigma'\end{aligned} \tag{OS-22}$$

$$\begin{aligned}&\langle \mathbf{raise\ irq}, \sigma\rangle \rightsquigarrow \sigma, \text{ where}\\ &irq <= IRQ_{MASK} \vee, IRQ \cap irq = \emptyset\end{aligned} \tag{OS-23}$$

$$\begin{aligned}&\langle \mathbf{enable_{irq}\ irqset}, \sigma\rangle \rightsquigarrow \sigma, \text{ at the same time}\\ &\langle enable_{irq}\ irqset, IRQ\rangle \rightsquigarrow IRQ \cup irqset\end{aligned} \tag{OS-24}$$

$$\begin{aligned}&\langle \mathbf{disable_{irq}\ irqse}, \sigma\rangle \rightsquigarrow \sigma, \text{ at the same time}\\ &\langle disable_{irq}\ irqse, IRQ\rangle \rightsquigarrow IRQ - irqset\end{aligned} \tag{OS-25}$$

The operational semantics shows the execution mechanism of the `xBIL` intermediate language. It is worth mentioning that the execution time and interrupt behaviours are taken into account in our framework. Users can lift their binary code with interrupt service routine involved to `xBIL` code directly, and simulate the interrupts by implementing the function $\iota(IRQ)$, which is mentioned previously themselves. In the next subsection, we will start to show the denotational semantics of `xBIL` program.

B. Denotational Semantics

The denotational semantics is an approach of formalising the meanings of programming languages by constructing mathematical objects (called denotations) that describe the meanings of expressions from the languages. The denotational semantics can help to bridge the program verification problem to a theorem proving problem. The denotations establish the logic relations of the objects in the discussion scope. Thus, giving the denotational semantics of `xBIL` is a necessary step for supporting the further analysis and verification work.

For describing the evaluation of `xBIL` expressions, we introduce the semantics function $\mathbb{E}$: $exp \to (\Sigma \to bvec)$. exp is the input expression, Σ identifies the set of all the program states and $bvec$ represents the set of bit-vectors. This function maps an expression to the pair composed by program state and evaluated result. The definition of function $\mathbb{E}$ is detailed in the following list:

$$\mathbb{E}[\![\mathbf{e_1}\ \diamond_{\mathbf{bvop}}\ \mathbf{e_2}]\!] = \lambda\sigma \in \mathbf{\Sigma}.(\mathbb{E}[\![\mathbf{e_1}]\!]\sigma \diamond_{\mathbf{bvop}} \mathbb{E}[\![\mathbf{e_2}]\!]\sigma) \tag{DS-1}$$

$$\mathbb{E}[\![\mathbf{number_{bv}}]\!] = \lambda\sigma \in \mathbf{\Sigma}.(\sigma_{\mathbb{R}}(\mathbf{reg_{id}})) \tag{DS-2}$$

$$\mathbb{E}[\![\mathbf{var_{id}}]\!] = \mathbb{E}[\![\mathbf{memory(addr, vt, e_t)}]\!] \tag{DS-3}$$

$$\mathbb{E}[\![\mathbf{mem_{id}[e]}]\!] = \mathbb{E}[\![\mathbf{memory(addr + n * offset, vt, e_t)}]\!] \tag{DS-4}$$

$$\begin{aligned}&\mathbb{E}[\![\mathbf{memory(exp, vt, e_t)}]\!] =\\ &\lambda\sigma \in \mathbf{\Sigma}.(\sigma_{\mathbb{M}}(\mathbf{F_{decN}(addr) + n - 1, \ldots, F_{decN}(addr)})),\\ &\text{where } e_t = little,\ n = bytes(vt),\ \mathbb{E}[\![exp]\!]\sigma = addr\end{aligned} \tag{DS-5}$$

$$\begin{aligned}&\mathbb{E}[\![\mathbf{memory(exp, vt, e_t)}]\!] =\\ &\lambda\sigma \in \mathbf{\Sigma}.(\sigma_{\mathbb{M}}(\mathbf{F_{decN}(addr), \ldots, F_{decN}(addr) + n - 1})),\\ &\text{where } e_t = big,\ n = bytes(vt),\ \mathbb{E}[\![exp]\!]\sigma = addr\end{aligned} \tag{DS-6}$$

The second part of the denotational semantics definition is regarding the `xBIL` commands. The command function $\mathbb{C}$:$command \to (\Sigma \to \Sigma)$ denotes the map from a given command to the states before and after the execution of this command.

$$\mathbb{C}[\![\mathbf{cmd_0; cmd_1}]\!] = \mathbb{C}[\![\mathbf{cmd_1}]\!] \circ \mathbb{C}[\![\mathbf{cmd_0}]\!] \tag{DS-7}$$

$$\mathbb{C}[\![\mathbf{reg_{id} := exp}]\!] = \lambda\sigma \in \mathbf{\Sigma}.(\sigma[\sigma_{\mathbb{R}}[\mathbb{E}[\![\mathbf{exp}]\!]\sigma/\mathbf{reg_{id}}]]) \tag{DS-8}$$

$$\begin{aligned}&\mathbb{C}[\![\mathbf{var_{id} := e_2}]\!] = \mathbb{C}[\![\mathbf{write(addr, v, vt, e_t)}]\!], \text{ where}\\ &var_{id} \downarrow_{address}\rightsquigarrow addr, \mathbb{E}[\![e_2]\!]\sigma = v,\\ &var_{id} \downarrow_{endiant_t}\rightsquigarrow e_t, var_{id} \downarrow_{valtype}\rightsquigarrow vt\end{aligned} \tag{DS-9}$$

$$\begin{aligned}&\mathbb{C}[\![\mathbf{exp \triangleright command}]\!] =\\ &\begin{cases}\{(\sigma,\sigma') \mid \sigma,\sigma' \in \Sigma \& (\sigma,\sigma') \in \mathbb{C}[\![command]\!]\}, & where\\ \langle exp, \sigma\rangle \rightsquigarrow v \wedge v \neq 0 & \\ \{(\sigma,\sigma) \mid \sigma \in \Sigma\}, & where\\ \langle exp, \sigma\rangle \rightsquigarrow v \wedge v = 0 & \end{cases}\end{aligned} \tag{DS-10}$$

$$\mathbb{C}[\![\mathbf{halt}]\!] = (\sigma, \sigma_{\mathbf{halt}})\sigma \in \mathbf{\Sigma} \tag{DS-11}$$

$$\begin{aligned}&\mathbb{C}[\![\mathbf{mem_{id}[e_1] := e_2}]\!] \equiv\\ &\mathbb{C}[\![\mathbf{write(addr + F_{decN}(offset) \times n, v, vt, e_t)}]\!], \text{ where}\\ &\mathbb{E}[\![e_1]\!]\sigma = offset, mem_{id} \downarrow_{endiant_t}\rightsquigarrow e_t, bytes(vt) = n,\\ &\mathbb{E}[\![e_2]\!]\sigma = v, mem_{id} \downarrow_{address}\rightsquigarrow addr, mem_{id} \downarrow_{valtype}\rightsquigarrow vt\end{aligned} \tag{DS-12}$$

For describing the time in `xBIL` program, we employ function $\mathbb{T}$:$command \to (\{T_{total}\} \to \{T_{total}\})$ to denote the change of the total running time before and after the execution of $command$.

$$\begin{aligned}&\mathbb{C}[\![\mathbf{cost\ n}]\!] = \lambda\sigma \in \mathbf{\Sigma}.(\sigma)\\ &\mathbb{T}[\![\mathbf{cost\ n}]\!] = \lambda\mathbf{T_{total}} \in \{\mathbf{T_{total}}\}.(\mathbf{T_{total} + n})\end{aligned} \tag{DS-13}$$

$$\begin{aligned}&\mathbb{C}[\![\mathbf{write(exp_1, exp_2, \tau_{reg}, e_t)}]\!] =\\ &\begin{cases}\lambda\sigma \in \Sigma.\sigma[\sigma_{\mathbb{M}}[mbv/F_{decN}(addr), \ldots & \\ \qquad\qquad F_{decN}(addr) + n]] & e_t = little\\ \lambda\sigma \in \Sigma.\sigma[\sigma_{\mathbb{M}}[mbv/F_{decN}(addr) + n, \ldots & \\ \qquad\qquad F_{decN}(addr)]] & e_t = big\end{cases}\\ &\mathbb{E}[\![exp_1]\!]\sigma = addr, bytes(\tau_{reg}) = n, \mathbb{E}[\![exp_2]\!]\sigma = mbv\end{aligned} \tag{DS-14}$$

$$\begin{aligned}&\mathbb{C}[\![\mathbf{jmp\ addr}]\!] = \mathbb{C}[\![\mathbf{\Pi[addr]}]\!], \text{ or}\\ &\mathbb{C}[\![\mathbf{jmp\ label_{id}}]\!] = \mathbb{C}[\![\mathbf{\Pi[\mathbb{L}(label_{id})]}]\!]\end{aligned} \tag{DS-15}$$

$$\begin{aligned}&\mathbb{C}[\![\mathbf{checkirq}]\!] = \mathbb{C}[\![\mathbf{\Pi[IVT(IRQ_{MASK})]}]\!], \text{ where}\\ &\iota(IRQ) \rightsquigarrow IRQ'_{MASK},\ IRQ'_{MASK} > IRQ_{MASK} \neq 0\end{aligned} \tag{DS-16}$$

$$\begin{aligned}&\mathbb{C}[\![\mathbf{checkirq}]\!] = \lambda\sigma \in \mathbf{\Sigma}.\sigma, \text{ where}\\ &\iota(IRQ) \rightsquigarrow IRQ'_{MASK},\ 0 \leq IRQ'_{MASK} \leq IRQ_{MASK}\end{aligned} \tag{DS-17}$$

$$\mathbb{C}[\![\textbf{raise irq}]\!] = \mathbb{C}[\![\mathbf{\Pi}[\mathbf{IVT}(\mathbf{irq})]]\!], \text{ where} \quad irq > IRQ_{MASK} \wedge IRQ \cap irq \neq \emptyset \tag{DS-18}$$

$$\mathbb{C}[\![\textbf{raise irq}]\!] = \lambda\sigma \in \mathbf{\Sigma}.\sigma, \text{ where} \quad irq \leq IRQ_{MASK} \vee IRQ \cap irq = \emptyset \tag{DS-19}$$

The interrupt semantics function $\mathbb{I}$:*command*$\rightarrow(\{IRQ\} \rightarrow \{IRQ\})$ is used to represent the change of the enabled interrupts set before and after the execution of *command.*

$$\mathbb{C}[\![\textbf{enable}_{\textbf{irq}} \textbf{ irqset}]\!] = \lambda\sigma \in \mathbf{\Sigma}.\sigma \quad \mathbb{I}[\![\textbf{enable}_{\textbf{irq}} \textbf{ irqset}]\!] = \lambda\mathbf{IRQ} \in \{\mathbf{IRQ}\}.\mathbf{IRQ} \cup \mathbf{irqset} \tag{DS-20}$$

$$\mathbb{C}[\![\textbf{disable}_{\textbf{irq}} \textbf{ irqset}]\!] = \lambda\sigma \in \mathbf{\Sigma}.\sigma \quad \mathbb{I}[\![\textbf{disable}_{\textbf{irq}} \textbf{ irqset}]\!] = \lambda\mathbf{IRQ} \in \{\mathbf{IRQ}\}.\mathbf{IRQ} - \mathbf{irqset} \tag{DS-21}$$

As of now, we have given all the semantics rules for both the operational and denotational semantics. Meanwhile, the virtual machine of `xBIL` called `xBVM` has been developed to simulate the binary code by executing `xBIL` program. From the definition of semantics, we can find that the bit-vector operation is quite often used. In the next section, we will focus our discussion on the `xBIL` bit-vector arithmetic and the corresponding decision procedure.

IV. Bit-Vector Arithmetic and Decision Procedures of xBIL Program

A computer system uses bit-vectors to encode information, for example numbers. Owing to the finite domain of these bit-vectors, the semantics of operations such as addition no longer matches what we are used to when reasoning about unbounded types, for example, the natural numbers. The range of the values is small, thus, the normal arithmetic operation may cause overflow on bit-vectors. To tackle the problems caused by the special features of bit-vector, the specific arithmetic is engaged in the calculation of bit-vectors. For enhancing the flexibility of our framework, we propose an approach to defining the bit-vector operators and give the corresponding decision procedure to solve the bit-vector formulas written following the rules we proposed. In this section, we firstly present the `xBIL` bit-vector and the operators of `xBIL` bit-vector arithmetic. Meanwhile, we will talk about the details regarding the extension of these operators. Afterwards, the decision procedure for `xBIL` bit-vector arithmetic formula will be the focus.

A. *Bit-vector Arithmetic*

Bit-vectors is used to encode information, for example, numbers. The 8-bits bit-vector $\langle 11001000\rangle$ can be used to represent the unsigned number 200. On the contrary, this bit-vector can also be the signed number -56. The decoding mechanism of the bit-vector leads to different outputs. In this part, firstly, we present the concepts regarding bit-vector. Secondly, the encoding/decoding will be discussed. In the end, all the commonly used operators regarding the `xBIL` bit-vector arithmetic are represented. We will also talk about the extension of these operators.

1) xBIL bit-vector and arithmetic: Bit-Vector b is a vector of bits with a given length l (or dimension, we use the same concept definition of bit-vector as in [12]). The formal definition to bit-vector is given below:

$$b : \{0, \ldots, l-1\} \rightarrow \{0, 1\}$$

The set of all 2^l bit-vectors of length l is denoted by $bvec_l$. The i-th bit of the bit-vector b is denoted by b_i. At the same time, global memory bit-vector is defined as $\mathbb{M}_b : \{0, \ldots, n-1\} \rightarrow \{0, 1\}$, where n denotes the max bit number that can be addressed in a special execution environment of xBIL program.

The syntax of the bit-vector arithmetic is given as follows:

$$bv \ ::= \ bv\ bvop\ bv \ \mid \ reg_{id} \ \mid \ memory(bv, \tau_{reg}, e_t) \ \mid \ Op_{enc}(number) \ \mid \ mem_{id}[F_{decN}(bv)]$$

bv bvop bv is an `xBIL` bit-vector which is calculated by performing the operation *bvop* on the given two bit-vectors. reg_{id} indicates a bit-vector represents the value of a given register. $memory(F_{decN}(bv), \tau_{reg}, e_t)$ is a bit-vector which represents the $\tau_{reg} \times 8$ bits length content of memory starting from $F_{decN}(bv)$. This bit-vector follows the endian format e_t. $mem_{id}[F_{decN}(bv)]$ represents a bit-vector denoted by offsetting $F_{decN}(bv)$ bytes from the start address of the memory block mem_{id}. $Op_{enc}(number)$ represents either the encoding or decoding operator.

2) Encoding and Decoding: The relation between the bit-vector and the corresponding information it represents is established by encoding and decoding mechanism. Before introducing the concepts of encoding and decoding, we present the λ-Notation for expressing the functions of encoding, decoding and other arithmetic operations.
λ-Notation is used to define functions. λ-Notation lists the context of a function and gives the corresponding result of these inputs. For instance, a lambda expression for a bit-vector with l bits can be expressed as below:

$$x_{[l]} = \lambda i \ \in \ \{0, \ldots, l-1\}.0$$

The expression above depicts the bit-vector by constructing a function mapping the bit index to its value. In this case, the 0 after $\lambda i \ \in \ \{0, \ldots, l-1\}$ represents that for each input i, the result is 0. More details regarding the λ-Notation can be found in [13].

The encoding and decoding operations are the functions links the bit-vector and the actual value. According to different encoding, even the same values can be encoded to different bit-vectors. Formally, the encoding function can be described as follows:

$$F_{encode} := number \ \rightarrow \ bvec_{[l]}$$

Similarly, the decoding function can be defined using the following form:

$$F_{decode} := bvec_{[l]} \ \rightarrow \ number$$

In previous sections, we employed F_{decN} and F_{encN} to denote the decode and encode function for unsigned integers. The definition of F_{decN} and F_{encN} are listed below:

$$F_{encN[l]}(n) = \lambda \ 0 \leq i < l.(n/2^{i+1} \ mod \ 2)$$
$$F_{decN}(b_{[l]}) = \sum_{i=0}^{l-1} b_i \times \ 2^i$$

For instance, $F_{encN[8]}(255)$ equals $\langle 11111111 \rangle$. On the contrary, $F_{decN[8]}(\langle 11111111 \rangle)$ can be evaluated to 255. If one specific bit-vector represents a float point number, it is necessary to involve another decode function for performing the right translation.

The endian format is also an important factor which should be taken into account while performing the encoding or decoding operations. For a given bit-vector b, the little and big endian format results can be defined as below:

$$\langle \ b \ \rangle_{little} := \sum_{i=0}^{l-1} b_i \times \ 2^i$$
$$\langle \ b \ \rangle_{big} := \sum_{n=0}^{(l \ div \ 8)-1} [(\sum_{i=8k}^{8k+7} b_i \cdot 2^{i \ mod \ 8}) \cdot 2^{8[(l \ div \ 8)-1-k]}]$$

3) Arithmetic Operations: `xBIL` bit-vector arithmetic operation definitions only use the basic integer and boolean arithmetic operations as well as the encoding or decoding functions we mentioned previously to describe the computation mechanism for each operator. In the remainder part of this subsection, we will list all the definitions of commonly used operators. These operators can be categorised as bit-capture, combination, bit-logic and normal arithmetic operators according to the usages.

$$\mathbf{BCO} : (\mathbf{bvec_l} \times \mathbf{N}) \rightarrow \mathbf{bvec_1} \tag{BCO-1}$$

The function BCO captures the i-th bit of the given bit-vector $bvec_l$. For a bit-vector x, we use x_i to express the same purpose for simplicity.

$$\mathbf{x} \succ_{(\mathbf{s},\mathbf{e})} =_{\mathbf{def}} \lambda(\mathbf{s} \leq \mathbf{i} \leq \mathbf{e}).\mathbf{x_i} \tag{BCO-2}$$

This function $\succ$ captures a range of bits from the original bit-vector x. The range starts from s and ends at e.

$$\mathbf{x_{[l]}} \circ \mathbf{y_{[k]}} =_{\mathbf{def}}$$
$$\lambda(0 \leq i \leq l+k-1). \begin{cases} x_i & (0 \leq i \leq l-1) \\ y_{(i-l+1)} & (l-1 \leq i \leq l+k-1) \end{cases} \tag{BCO-3}$$

This function is the concat of two bit-vectors x and y. The length of returned new bit-vector is $l+k$.

$$\mathbf{BLO_{or}} : (\mathbf{bvec_1} \times \mathbf{bvec_1}) \rightarrow \mathbf{bvec_1} \tag{BLO-1}$$

This rule calculates the logic or result of two given 1-bit bit-vectors. Similarly, and and neg present the logic and negation of bit-vectors. For simplicity, we use $\neg x$ to denote the negation of bit-vector x.

$$\mathbf{BLO_{and}} : (\mathbf{bvec_1} \times \mathbf{bvec_1}) \rightarrow \mathbf{bvec_1} \tag{BLO-2}$$

$$\mathbf{BLO_{neg}} : (\mathbf{bvec_1}) \rightarrow \mathbf{bvec_1} \tag{BLO-3}$$

$$\mathbf{x} \times_{[\mathbf{2l}]} \mathbf{y} =_{\mathbf{def}}$$
$$\lambda(0 \leq i \leq 2l-1) \sum_{[2l]i=0}^{l-1} \begin{cases} \mathbf{F_{encN[l]}} \circ \mathbf{x_{[l]}} \ll \mathbf{i} & x_i \\ \mathbf{F_{encN[2l]}}(\mathbf{0}) \ll \mathbf{i} & \neg x_i \end{cases} \tag{AR-1}$$

$$\mathbf{x} +_{[\mathbf{l}]} \mathbf{y} =_{\mathbf{def}} \lambda(\mathbf{0} \leq \mathbf{i} \leq \mathbf{l}-\mathbf{1}).$$
$$\begin{cases} (x_i \oplus y_i \oplus ((x_{i-1} \wedge y_{i-1}) \vee (\bigcup_{k=0}^{i-2} \\ (\bigwedge_{j=k+1}^{i-1}(x_j \oplus y_i) \wedge x_k \wedge y_k)))) & (2 \leq i \leq l-1) \\ (x_i \oplus y_i \oplus (x_{i-1} \wedge y_{i-1})) & (i=1) \\ x_i \oplus y_i & (i=0) \end{cases} \tag{AR-2}$$

$+_{[l]}$ operator adds two bit-vectors by using the logic $\oplus$ and $\wedge$ operations. This definition of $+_{[}l]$ operator can handle the overflow and give a certain result as in practical.

$$x -_{[l]} y =_{def} \lambda(0 \leq i \leq l-1).$$
$$\begin{cases} (x_i \oplus y_i \oplus ((\neg x_{i-1} \wedge y_{i-1}) \vee (\bigcup_{k=0}^{i-2} \\ (\bigwedge_{j=k+1}^{i-1}(x_j \odot y_i) \wedge \neg x_k \wedge y_k)))) & (2 \leq i \leq l-1) \\ ((x_i \oplus y_i) \wedge (x_{i-1} \vee \neg y_{i-1})) \vee \\ ((x_i \odot y_i) \wedge (\neg x_{i-1} \wedge y_{i-1})) & (i=1) \\ x_i \oplus y_i & (i=0) \end{cases} \tag{AR-3}$$

$$\mathbf{x} \ \&_{[\mathbf{l}]} \ \mathbf{y} \ =_{\mathbf{def}} \ \lambda(\mathbf{0} \leq \mathbf{i} \leq \mathbf{l}-\mathbf{1}).(\mathbf{x_i} \ \wedge \ \mathbf{y_i}) \tag{AR-4}$$

$$\mathbf{x} \ |_{[\mathbf{l}]} \ \mathbf{y} \ =_{\mathbf{def}} \ \lambda(\mathbf{0} \leq \mathbf{i} \leq \mathbf{l}-\mathbf{1}).(\mathbf{x_i} \ \vee \ \mathbf{y_i}) \tag{AR-5}$$

Rule AR-4 and AR-5 perform logic $\wedge$ and vee operations for each pair of corresponding bits in bit-vector x and y.

$$\mathbf{x_{[l]}} \ll \mathbf{y_{[k]}} =_{\mathbf{def}} \lambda(\mathbf{0} \leq \mathbf{i} \leq \mathbf{l}-\mathbf{1}).$$
$$(\bigvee_{s=0}^{l-1} \begin{cases} (x_{i-s} \wedge (y =_{[k]} F_{encN[k]}(s)))) & (i \geq s) \\ 0 (i < s) \end{cases} \tag{AR-6}$$

$$\mathbf{x_{[l]}} \gg \mathbf{y_{[k]}} =_{\mathbf{def}} \lambda(\mathbf{0} \leq \mathbf{i} \leq \mathbf{l}-\mathbf{1}).$$
$$(\bigvee_{s=0}^{l-1} \begin{cases} (x_{i+s} \wedge (y =_{[k]} F_{encN[k]}(s)))) \ (i+s \leq l) \\ 0 \ (i+s > l) \end{cases} \tag{AR-7}$$

$\ll$ performs $F_{decN[k]}(y)$ bits left-shift on bit-vector x. On the contrary, $\gg$ behaves oppositely.

$$\mathbf{x} \odot_{[\mathbf{l}]} \mathbf{y} =_{\mathbf{def}} \lambda(\mathbf{0} \leq \mathbf{i} \leq \mathbf{l}).((\mathbf{x_i} \wedge \mathbf{y_i}) \vee (\neg \mathbf{x_i} \wedge \neg \mathbf{y_i})) \tag{AR-8}$$

$$\mathbf{x} \oplus_{[\mathbf{l}]} \mathbf{y} =_{\mathbf{def}} \lambda(\mathbf{0} \leq \mathbf{i} \leq \mathbf{l}).((\mathbf{x_i} \wedge \neg \mathbf{y_i}) \vee (\neg \mathbf{x_i} \wedge \mathbf{y_i})) \tag{AR-9}$$

$$\mathbf{x} \vee_{[1]} \mathbf{y} =_{\mathbf{def}} \mathbf{x_0} \vee \mathbf{y_0} \tag{AR-10}$$

$$\mathbf{x} \wedge_{[1]} =_{\mathbf{def}} \mathbf{x_0} \wedge \mathbf{y_0} \tag{AR-11}$$

$$\mathbf{x} =_{[l]} \mathbf{y} =_{\mathbf{def}} \lambda. \bigwedge_{\mathbf{i=0}}^{\mathbf{l-1}} ((\neg \mathbf{x_i} \vee \mathbf{y_i}) \wedge (\neg \mathbf{y_i} \vee \mathbf{x_i})) \tag{AR-12}$$

$$\mathbf{x} \neq_{[l]} \mathbf{y} =_{\mathbf{def}} \lambda. \bigvee_{\mathbf{i=0}}^{\mathbf{l-1}} ((\mathbf{x_i} \wedge \neg \mathbf{y_i}) \vee (\mathbf{y_i} \wedge \neg \mathbf{x_i})) \tag{AR-13}$$

$$\mathbf{x} >_{[l]} \mathbf{y} =_{\mathbf{def}} \lambda(\mathbf{0} \leq \mathbf{i} \leq \mathbf{l-1}). \bigcup_{i=0}^{l-1} (\bigwedge_{k=i+1}^{l-1} (x_k =_{[l]} y_k) \wedge x_i \wedge \neg y_i) \tag{AR-14}$$

$$\mathbf{x} <_{[l]} \mathbf{y} =_{\mathbf{def}} \lambda(\mathbf{0} \leq \mathbf{i} \leq \mathbf{l-1}). \bigcup_{i=0}^{l-1} (\bigwedge_{k=i+1}^{l-1} (x_k =_{[l]} y_k) \wedge \neg x_i \wedge y_i) \tag{AR-15}$$

$\odot_{[l]}$ is the XNOR operation. $\oplus_{[l]}$ represents the XOR operation. $\vee_{[1]}$ performs OR operation on the given bit-vectors. $\wedge[1]$ is the logic AND. $=_{[l]}$ returns $la1ra$ if the bit-vectors x and y are equal. $\neq_{[l]}$ is opposite to $=_{[l]}$. $>_{[l]}$ is $la1ra$ if x is greater than y for each bit. $<_{[l]}$ returns opposite result to $>_{[l]}$.

B. Decision Procedure of xBIL Bit-Vector Formula

The decision procedure of bit-vector is a major way to check whether bit-vector formula is satisfied or not. The bit-vector formula depicts the logic relations of the given bit-vectors. In one specific bit-vector formula, every bit of bit-vectors is represented by dedicated boolean variable. In this case, the bit-vector solving problem can be transformed to a SAT solving problem. There are two mainstream decision procedures as of now, Flattening [14] and Incremental-Flattening [15]. We extend these two approaches and propose a new algorithm for adapting the solution to `xBIL` bit-vector formula solving problem. In this part, we introduce the `xBIL` bit-vector formula first. Next, we demonstrate the algorithm we proposed.

1) Formula of xBIL Bit-vector: In this part, we will introduce the basic concepts of `xBIL` bit-vector formula, and demonstrate how to use xBIL bit-vector formula and formula group to indicate an xBIL specification. First of all, the syntax of xBIL formula is defined as follows:

$$\begin{aligned} formula_{bv} &: formula_{bv} \vee formula_{bv} \mid \neg formula_{bv} \\ &\mid (formula_{bv}) \mid atom_{bv} \\ atom_{bv} &: bvop_{[1]}\ bv \mid bv\ bvop_{[1]}\ bv \mid true \mid false \end{aligned}$$

The term $atom_{bv}$ is an atom proposition which represents a 1-bit bit-vector that can be calculated by performing bit-vector operations on one or two normal bit-vectors. The only constraint of $atom_{bv}$ is that the corresponding bit-vector has to be evaluated to a 1-bit bit-vector. If $atom_{bv}$ represents bit-vector $\langle 1 \rangle$, it is $true$. Otherwise, it is $false$. $formula_{bv}$ is composed by $atom_{bv}$ and logic OR as well as NOT connectors.

For instance the formula $x +_{[8]} y = F_{encN[8]}(44)$ represents that there are two 8-bits bit-vectors x and y, the sum of them is the bit-vector by encoding the unsigned integer 44. Additionally, if we add two constraints to limit the value of x and y within 100, then we can get the following formula: $x +_{[8]} y = F_{encN[8]}(44) \wedge x_{[8]} <_{[8]} F_{encN[8]}(100) \wedge y_{[8]} <_{[8]} F_{encN[8]}(100)$. These two cases show very simple cases to identify the specification on `xBIL` bit-vectors. In the next part, we will detail the algorithm to solve these constraints and give the answers to these formulas.

2) Algorithm: `xBIL` bit-vector decision procedure applies hierarchy decision method by dividing large proposition formulas into small ones. There are some concepts and structures when we introduce our algorithm. $e(\phi)$ indicates the skeleton of the formula. $e(\phi)$ can be computed by replacing the certain boolean variable groups with the other irrelevant boolean variables directly. $BV(\phi)$ is the set of all the bit-vectors in formula ϕ. $At(\phi)$ is the set of all the `xBIL` atom bit-vectors in formula ϕ. For each bit-vector in the atomic propositions, $e(bv)$ denotes the final simplified boolean variables vector after applying the operation rules mentioned previously. $e(bv)_i$ means the i-th bit of $e(bv)$.

In Algorithm 1, we list all the steps for solving a given `xBIL` bit-vector formula. First of all, we make β equal to $e(\phi)$. Afterwards, supposing SAT_r is the result of β after SAT-SOLVER processed. If SAT_r is $unsatisfied$, then the algorithm returns $unsatisfied$ and terminates itself. If not, all the bits for each bit-vector bv in $BV(\phi)$ will be replaced by the new boolean variables getting from $e(bv)$. Now, we have used the boolean variables to represent the bit-vectors in the given formula. Then, we iterate every `xBIL` atom bit-vector α in ϕ. Next, we calculate the constraint logic sub-formula by applying bit-vector operation rules to α. Afterwards, the value of β will be the conjunction of β itself and the calculated sub-formula BV-Compute(α). We simplify the formula β to reduce the size before sending it to the SAT solver. Supposing SAT_r is the result of the SAT-SOLVER with input β. If SAT_r is $unsatisfied$, then returns $unsatisfied$. Otherwise, continues the iteration. The iteration will be accomplished and returns $unsatisfied$ when there is no more atom bit-vector need to be processed.

Compared with the existing decision procedure, especially the Flattening and Incremental-Flattening algorithms, there are some advantages of the algorithm we proposed in this paper. Firstly, our algorithm is bits number independent. Other algorithms generate additional variables for the skeletons and the skeletons of sub-formulas. Meanwhile, the incremental algorithm uses some temporal variables for solving the formula.

Algorithm 1 xBIL BV Decision Procedure

```
1:  $\beta := e(\phi)$;
2:  $SAT_r := SAT - SOLVER(\beta)$;
3:  if $SAT_r$=$unsatisfied$ then
4:      return $unsatisfied$;
5:  else
6:      for each $bv_{[l]} \in BV(\phi)$ do
7:          for each $i \in \{0, \ldots, l-1\}$ do
8:              replace $e(bv)_i$ new boolean variable
9:          end for
10:     end for
11:     for each $\alpha \in At(\phi)$ do
12:         $\beta := \beta \wedge BV - Compute(\alpha)$;
13:         $\beta := Simpilify(\beta)$;
14:         $SAT_r := SAT - SOLVER(\beta)$;
15:         if $SAT_r$=$unsatisfied$ then
16:             return $unsatisfied$;
17:         end if
18:     end for
19:     return $satisfied$;
20: end if
```

Our framework has fixed numbers of boolean variables, this helps to handle large scale formulas. Secondly, some existing algorithms can only handle the specific operations as they use a variety of mathematics operators. Instead, in our approach, only encoding/decoding and logic operators are occupied. Our approach is similar to the implementation of a circuit, which implements the functions only using the basic logic gates. From this perspective, our method is flexible and expressive.

Back to the examples we mentioned previously. By applying the algorithm we listed above, we can get one solution model by using Z3 [16] as SAT solver for formula $x +_{[8]} y = F_{encN[8]}(44)$. The model we get is listed as follows:

```
(define-fun x7 () Bool false)    (define-fun y7 () Bool true)
(define-fun y1 () Bool false)    (define-fun y0 () Bool false)
(define-fun x2 () Bool false)    (define-fun x1 () Bool false)
(define-fun x6 () Bool true)     (define-fun y5 () Bool false)
(define-fun y6 () Bool true)     (define-fun y3 () Bool false)
(define-fun x3 () Bool true)     (define-fun y4 () Bool true)
(define-fun x5 () Bool false)    (define-fun y2 () Bool true)
(define-fun x4 () Bool true)     (define-fun x0 () Bool false)
```

From this solution, we can see x=$\langle 01011000 \rangle \rightsquigarrow 88$ and y=$\langle 11010100 \rangle \rightsquigarrow 212$ is one solution to formula $x +_{[8]} y = F_{encN[8]}(44)$. However, this solution is correct as the overflow is occurred. This case shows the power of xBIL bit-vector and its decision procedure as well. Analogously, send $x +_{[8]} y = F_{encN[8]}(44) \wedge x_{[8]} <_{[8]} F_{encN[8]}(100) \wedge y_{[8]} <_{[8]} F_{encN[8]}(100)$ to our algorithm, we can get x=20 and y=24 as the added constraints help to limit the range of these two bit-vectors.

When we use a decision procedure to prove the correctness of the program, we need to describe the specifications of the program. xBIL bit-vector formula is a very straightforward tool to express these specifications. For a given specification formula ϕ, usually, we need to get the negation of the formula $\neg\phi$. Then attempting to check whether this formula is satisfied. If there is a solution, that means the original formula ϕ is not forever true. In this case, we can confirm that at least part of the program is not implemented correctly.

V. The Application of xBIL

We have evaluated our language on the analysis and verification of several systems, including real-time operating system, fuel injection system and other real-time controlling systems. In these cases, xBIL code are lifted from the binary firmware directly. We specified the properties by using xBIL bit-vector formulas for some code segments, and employed RT-CTPL we proposed to depict the interrupt relevant temporal properties. In the following part, we introduce the applications of xBIL on the analysis and verification of a commercial real-time operating system.

We have applied our approach to the function and interrupt safety verification of a real-time commercial operating system - ORIENTAIS, which is developed by iSoft Infrastructure Software Co., Ltd, in China. Now, OSEK (Offene Systeme und deren Schnittstellen fr die Elektronik in Kraftfahrzeugen) is a standards body that has produced specifications for embedded operating system, a communications stack, and a network management protocol for automotive embedded systems [17]. OSEK was designed for providing a standard software architecture for the various electronic control units (ECUs) throughout the car. It is the most widely used automotive electronic oriented operating system standard all over the world. The ORIENTAIS is an OSEK standard based commercial real-time embedded system for the automotive industry. The binary comes from the firmware of one ECU built upon this system. Several potential interrupt relevant problems caused by memory access conflicts are detected by using our method. All these problems are confirmed by the vendor of ORIENTAIS.

The target operating system ORIENTAIS implements all the features of OSEK standard. In the very beginning, the firmware we used in this case is compiled for Freescale 9S12 micro-processor. However, this operating system targets most of the mainstream micro-controller chips in the automotive industry. We use unpacking and disassembly tool IDA Pro [18] for obtaining the assembly code and data segments of the given firmware. Afterwards, we use tools to parse the assembly code and static data blocks. The assembly code blocks are translated to xBIL instructions, and the static data blocks are noted in the context declaration block of xBIL program. In our evaluation work, we get xBIL code for multiple hardware platforms such as MPC5634, ARM-9 and ARM Cortex-M4. Although there are many different target platforms, we can still use a unified technique framework to handle the analysis and verification work as the code from all these platforms can be expressed as xBIL programs. Based on xBIL code, we performed data race analysis, memory access analysis, OS API function verification and interrupt safety properties verification.

We have found 35 potential problems in the early version of ORIENTAIS. With our help, The software vendor has fixed all the problems we found according to our verification report. The `ORIENTAIS` has been certified by `OSEK` standard group and deployed on over 1.38 million cars in China as of now.

VI. Conclusion

In this paper, we have presented a binary intermediate language towards embedded systems. This language can be used to describe the hardware runtime environment and instructions for a variety of platforms. Time and interrupt features have been taken into account in this language. The syntax and semantics were given in this paper. The detailed operational and denotational semantics provided the basic information for simulating the system and proving the specification. In the semantics of `xBIL` program, all the evaluation operations are bit-vector oriented. The definitions of bit-vector arithmetic operations have been presented in this paper as well. Meanwhile, we have talked about the bit-vector formula and its corresponding decision procedure. The application case showed the feasibility and flexibility of our language. Several bugs have been successfully detected for a commercial operating system based on our approach.

In the future, we plan to improve `xBIL` by adding new features for supporting modern processors. Additional plug-ins on the tool `xBVM` we built for simulating system `xBIL` program will be developed to support more architectures, especially the architectures for embedded platforms.

References

[1] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "Codesurfer/x86a platform for analyzing x86 executables," in *Compiler Construction*. Springer, 2005, pp. 250–254.

[2] A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps, "Directed proof generation for machine code," in *Computer Aided Verification*. Springer, 2010, pp. 288–305.

[3] "Microsoft Phoenix Framework." http://research.microsoft.com/phoenix/.

[4] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *Computer Aided Verification*. Springer, 2008, pp. 423–427.

[5] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *Information systems security*. Springer, 2008, pp. 1–25.

[6] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," DTIC Document, Tech. Rep., 2009.

[7] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs." in *NDSS*, 2011.

[8] T. Lundqvist and P. Stenström, "Integrating path and timing analysis using instruction-level simulation techniques," in *Languages, Compilers, and Tools for Embedded Systems*. Springer, 1998, pp. 1–15.

[9] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[10] J. Shi, J. He, H. Zhu, H. Fang, Y. Huang, and X. Zhang, "Orientais: Formal verified osek/vdx real-time operating system," in *Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on*. IEEE, 2012, pp. 293–301.

[11] J. Shi, L. Zhu, H. Fang, J. Guo, H. Zhu, and X. Ye, "xbil–a hardware resource oriented binary intermediate language," in *Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on*. IEEE, 2012, pp. 211–219.

[12] D. Kroening and O. Strichman, *Decision procedures*. Springer, 2008, vol. 5.

[13] D. H. Warren, *Higher-order extensions to Prolog-are they needed*. Department of Artificial Intelligence, University of Edinburgh, 1981.

[14] C. M. Wintersteiger, Y. Hamadi, and L. De Moura, "Efficiently solving quantified bit-vector formulas," *Formal Methods in System Design*, vol. 42, no. 1, pp. 3–23, 2013.

[15] A. Myles and D. Zorin, "Global parametrization by incremental flattening," *ACM Transactions on Graphics (TOG)*, vol. 31, no. 4, p. 109, 2012.

[16] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[17] J. Lemieux, "The osek/vdx standard: Operating system and communication," *Embedded Systems Programming*, vol. 13, no. 3, pp. 90–109, 2000.

[18] C. Eagle, *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2008.