Verifying Interrupt Properties of Real-Time Embedded System in Binary Code Perspective

Jianqi Shi*, Huixing Fang†, Yanhong Huang*, Fu Song†, Jian Guo†
*National Trusted Embedded Software Engineering Technology Research Center,
East China Normal University, China
†Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China
Email:{*jqshi,†wxfang,†jguo,†fsong,*hyhuang}@sei.ecnu.edu.cn

***Abstract*—The interrupt mechanism is indispensable in embedded software due to its many benefits such as switching context and enhancing the performance. In this scenario, the traditional approaches to guaranteeing the correctness of software are not efficient enough. Having interrupt involved, the complicated and non-deterministic environment should be taken into account for each verification phase. In this paper, we propose a novel approach to verify the interrupt relevant properties from the low-level binary code perspective. At first, the control flow graphs (`CFGs`) are retrieved from `xBIL` [1] with the time and interrupt properties reserved. The `xBIL` is a binary intermediate language we proposed to represent the machine instructions crossing various hardware architectures. Meanwhile, the data flow graphs (`DFGs`) are built and analysed for determining the relations between registers and the values for particular memory addresses. We present an automatic approach to construct the Labelled Transition System (`LTS`) by combining the `CFG` and `DFG` models. Next, the program slicing technique is applied to reduce the program model scale. Finally, users can use `RT-CTPL` we proposed to identify the safety properties for performing the security and quantitative analysis. To prove the feasibility of our approach, we apply our method to the verification of a commercial automotive operating system (`ORIENTAIS` [2]), Fuel Injection System and Mechanical Controlling System. All of these prove that our approach brings great help to the development of real-time embedded software.**

***Index Terms*—Binary Code, Quantitative Analysis, Verification, Interrupt, Safety Property**

I. Introduction

The interrupt mechanism is indispensable in embedded systems due to lots of factors, such as switching context and enhancing the performance. It is one of the most commonly used techniques for embedded systems, especially real-time operating systems. Most embedded systems are connected to multiple external devices, and the interrupt mechanism is introduced as the interface between the controlling program (or operating system kernel) and devices, which guarantee the system to interact with its environment in a timely manner. The interrupt requests are usually implemented in terms of asynchronous signals (i.e.,`POSIX` [3] style signals) or events. As depicted in Fig.1, when an interrupt request arrives, the processor attempts to suspend the current running program, and switch to the corresponding interrupt handling program by locating the entry address in the interrupt vector table. The interrupt requests are implemented in terms of signals or events which can be triggered by either programs or external devices, in other words, by the system or the environment. Afterwards, the code that has already been associated with the arriving interrupt starts to run. Finally, the control point returns back to the place at which the interrupt occurred and the preserved execution context should be recovered to the status before the interrupt happened. It deserves to be specially noted that some systems allow the executing interrupt handling program to be preempted by another one with higher priority. Interrupt significantly improves system performance, but it can raise new potential problems. A great number of interrupts may completely stall the system under extreme conditions. The safety of program with interrupts becomes one key ingredient of program correctness. However, the non-determinism of interrupt occurrence makes the analysis and verification of real-time embedded systems much more difficult. Thus, the traditional way to ensure program correctness is not applicable in this scenario.

With more and more software applications being used in critical infrastructures, there is a rising requirement for enhancing the security of embedded software. This rising requirement brings challenges to the verification of interrupt involved programs. The first challenge of verifying interrupt relevant properties is providing a platform-independent framework to handle the binary firmware of the embedded system. As most of the vendors do not provide the source code of the firmwares, the analysis has to be performed on the binary code level directly. However, the variety of hardware platforms makes the analysis work very complicated. The second problem is caused by the expressiveness and usability of the interrupt property. Traditional approaches can only deal with the normal temporal property or the `WCET` [4], but the predicate and arithmetic relation are also desired in pracital usage scenario. These absent reqiured features will help users to describe the constraints in a more precise way and simplify the property expression.

In this paper, we present a novel approach to evaluate and verify the interrupt properties of real-time embedded systems. Firstly, we lift the binary code of firmware to an intermediate language we proposed. Afterwards, the `CFGs` and `DFsG` are constructed by analysing the lifted intermediate language. `CFGs` and `DFGs` are combined as `LTS` model. In the end, the

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

combined LTS model and predefined properties specified by the logic formula we proposed are sent to the model checker for identifying the potential problems of the binary program.

The approach we proposed brings new benefits to existing development work. These benefits can be summarised as follows:

a) **Model checking the interrupt properties in binary code level**. Most of the embedded software products are not open-source. Researchers have neither source nor documents for study. To analyse real-time embedded applications, it is challenging to reconstruct the program model using the reverse engineering technique. The approach we proposed can verify interrupt properties of real-time embedded systems from the binary code perspective. This helps firmware vendors to evaluate the quality of their products in the final phase. Another benefit of the binary code level verification of real-time embedded system is assisting the integrators in checking the required features or properties of one specific software components provided by other developers.

b) **The proposed approach is platform independent**. Compared with other program analysis work in binary code level, our approach proposed an intermediate language to support more hardware platforms. Taking the advantage of this feature, vendors can perform the verification for multiple binary platform targets within one framework.

c) **Verifying properties based upon the predicates**. In order to handle variables in safety properties of both main and interrupt handling programs, we introduce the RT−CTPL (Real-Time Computation Tree Predicate Logic) which is an extended version of CTPL. The CTPL supports predicates over the CTL formulas. Users can use quantifiers to express the specifications. With the help of quantifiers, the property formula can be simplified without declaring the property of one given variable dedicatedly. The real-time CTPL involves time cost and interrupt operators. These operators may conduce towards describing the interrupt properties in a simpler and more precise way.

The rest of this paper is organised as follows. In Section II, we will overview the approach we proposed in this paper. The details of the whole checking process will be described. Afterwards, we introduce the intermediate language we proposed with some examples in Section III. Section IV includes the content regarding modelling the xBIL program by constructing CFG, DFG and the combined LTS model. In this section, we show the method of abstracting the final LTS model with program slicing technique as well. The SE−CTPL and interrupt safety properties are introduced in Section V. After that, the verification algorithm is discussed in the remainder part of this section. We evaluate our approach by demonstrating three cases in Section VI. All the related work are listed in Section VII. At the end, in Section VIII we conclude the paper.

II. Method Overview

In this section, we overview the approach proposed in this paper. In order to verify the interrupt safety properties from the binary code perspective, we employ a variety of models and approaches such as xBIL [1], CFG [5], DFG [6], LTS and RT−CTPL. Fig.2 illustrates the whole checking process of our method. Moreover, the details of Fig.2 are given in the remainder of this section.

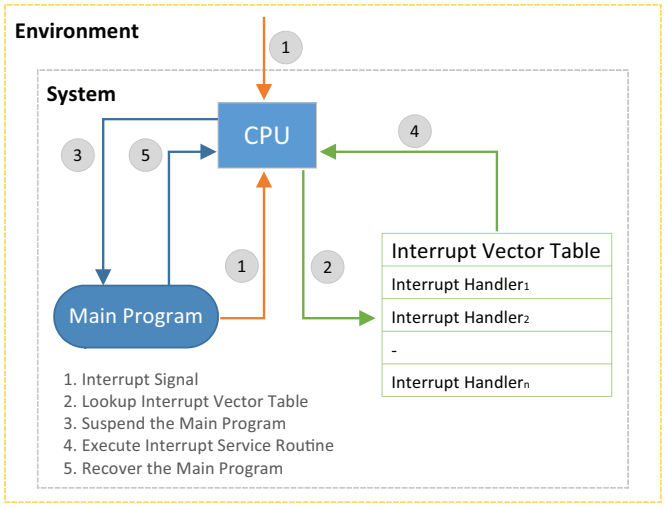

Fig. 1. The Interrupt Mechanism

In the first move, we extract the xBIL instructions from the Embedded System Firmware. For analysing Real-Time Embedded System in binary code level and supporting variety of hardware architectures, we proposed xBIL as the binary code intermediate language. xBIL can be directly constructed from a group of assembly code, exported symbol table and interrupt vector table. These resources can be obtained from the output of the tool IDA Pro. IDA Pro [7] is one of the most commonly used tool for reversing engineering. We pick the target firmware as the input of IDA Pro and collect the disassembled code and symbol segments as output. Afterwards, the output are sent as the input of our tool xBVM. xBVM is the tool we built to generate the corresponding xBIL program based upon the resources we mentioned previously. xBVM generates multiple code segments according to the input resources, including the main program, exported API functions and interrupt service routines (ISRs). These segments will serve the program modelling in next phase. As of now, the xBVM only supports ARM, x86, PowerPC and 9S1x series instruction set. Obviously, as xBIL is a cross-platform intermediate language, more processor architectures can be easily supported by adding new plugins to xBVM in the future.

In the second phase, firstly, we build the CFGs and DFGs models of the xBIL programs. In the very beginning, following the definition of xBIL program CFG model, CFGs of the xBIL programs are constructed. Next, the CFGs of main program and ISRs are combined together. We proposed the lock area concept to assist the building operations of the final merged CFG. The final model comprises both the main program and all the ISRs which may be triggered during the runtime of main program. We can get the DFG models analogously except the analysis of the relations between the objects, such as registers and the memory addresses. Next, the CFG and DFG models are combined as a more comprehensive

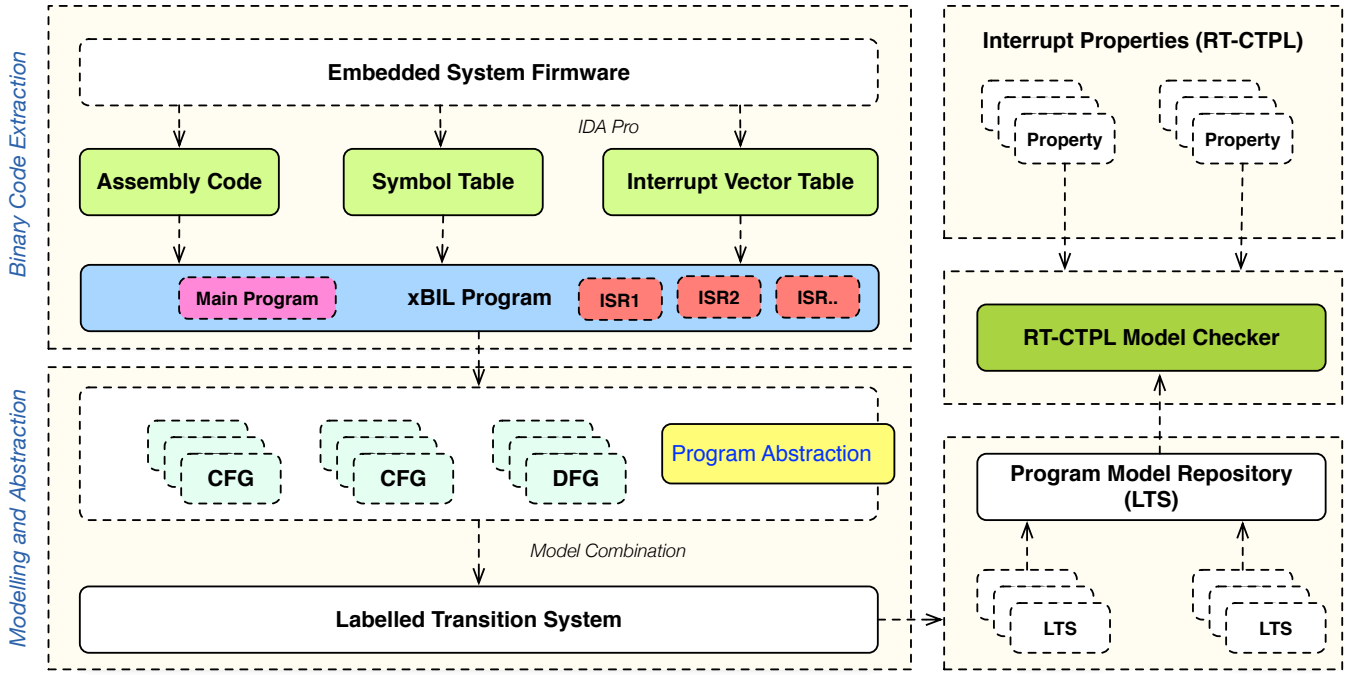

Fig. 2. Method Overview

LTS model using the rules we proposed. The LTS model includes both the control flow and data relation information which are required by the model checking of interrupt properties. Considering the model scale and properties need to be checked, the constructed LTS model should be abstracted to avoid the state explosion. To achieve this, we applied the program slicing technique. The nodes irrelevant with the properties should be removed from this model.

After we build the program model, we employ the RT−CTPL to specify the interrupt safety properties. RT−CTPL is an extended version of CTPL [8] which has been successfully applied for malware detection. The good support of predicate makes the CTPL becoming one of the most efficient approaches to illustrating the binary program behaviours. However, it is not the best choice for programs with interrupt due to the lack of operators regarding the time counting and interrupt behaviours. Taking the advantage of RT−CTPL, we are able to specify the time and interrupt relevant properties by using the RT−CTPL formulas. All the built LTSs are stored in the program model repository. Afterwards, the interrupt properties and the corresponding LTS models are sent to the RT−CTPL model checker for verification. The model checker checks every property for each LTS model. If the property is not satisfied, the corresponding counter examples will be given.

Most of the steps in our method are automated. Meanwhile, the tool xBVM can be used as the simulator for analysing and locating the detected problems as well. In the following sections, the details will be presented for each phase we mentioned in this part.

III. The xBIL Language

In this section, we introduce the binary intermediate language (xBIL) we applied in this paper. As more and more hardware platforms are brought out, the instruction set architecture (ISA) becomes very complicated. The binary intermediate language fills the gap between the universal instruction set and the particular hardware platform. Nonetheless, most of the binary intermediate languages only focus on the x86 platform while ignoring the embedded systems. Although, minority intermediate language such as LLVM can provide more general way to express the programs over multiple architectures, however, this feature only directs at high level languages. These intermediate language cannot manipulate the low level code just like assembly code. They only reserve the assembly code as special function segment. In this part, we firstly list the details of the xBIL language. Afterwards, we discuss the approach of extracting the xBIL from the embedded system firmware.

A. Language Overview

The xBIL is a binary intermediate language we proposed to represent the machine instructions for multiple hardware platforms. It is designed to illustrate the ISA in a universal way. This design is embedded system friendly, and it takes both the instruction execution time and interrupt behaviours into account as well. Meanwhile, another key feature of xBIL plays an important role to describe hardware relevant operations. The xBIL is a hardware resource oriented Binary Intermediate Language. Hardware resource is one important concept in xBIL. It represents the components of particular hardware platforms, for instance, the bus, the instruction fetcher, etc. In xBIL, not only the instructions but also

the corresponding hardware resources are indicated in the statements. This helps to depict the resource competition relations such as the pipeline mechanism. More than that, the context information of the binary code program including register type, variable address and interrupt service routine can also be easily expressed in xBIL. These information is partially extracted from the target platform according to the corresponding specification and the export symbol table of a specific firmware.

We list the xBIL structure definition as follows:

```
program            ::=  atom;program
atom               ::=  addr,res_zone_actions
res_zone_actions   ::=  res_zone_action;res_zone_actions
res_zone_action    ::=  stmts:res_occ_time_set
stmts              ::=  stmt;stmts
res_occ_time_set   ::=  res_occ_time,res_occ_time_set
res_occ_time       ::=  res_id:time
```

In order to preserve the atomicity of the machine instructions, atoms are introduced to express the real machine instructions which cannot be interrupted. The xBIL program is just composed by a sequence of atoms. An atom comprises a sequence of resource zone actions which define the hardware resource occupation time and statements. When executing, two or more atoms can be activated at the same time. If an atom is activated, the corresponding resource zone actions in this atom attempt to require the hardware resource declared in this zone and perform the actions step by step. In case the required hardware resource is occupied by other resource zone action, the requiring action should be suspended until this resource is released. The detailed execution mechanism can be found in [1].

The behaviours of an atom are implemented as a sequence of statements declared in resource zone actions. The statements here are used to simulate the equivalence function of a real instruction. All the statements are instantaneous, the time cost of resource zone actions are calculated by the occupation time of hardware resources. The following list is the syntax of atom statements.

```
stmt ::= reg_id:=exp | var_id:=exp!τ_e | exp ▷ stmt | halt |
         write(exp_V, exp_A, τ_e,τ_reg) | mem_id[exp]:=exp!τ_e|
         act (label_id | addr) | clratoms |
         raise irq | enable{irq,...} | disable{irq,...}
```

We use $\texttt{reg}_{id}\texttt{:=exp}$ to indicate the assignment on the register. $\texttt{var}_{id}\texttt{:=exp!}\tau_e$ represents the variable assignment of memory following the τ_e endian format. exp ▷ stmt can be used to represent conditional execution of statement including the conditional jump. The stmt can only be executed when the value of exp equals 1 or true. halt terminates the program and no atom can be activated any more. write(exp_V, exp_A, τ_e,τ_{reg}) evaluates the value of exp_V and write it to the memory which located at the address represented by exp_A. τ_{reg} and τ_e are given for determining the value type and endian format. $\texttt{mem}_{id}\texttt{[exp]:=exp!}\tau_e$ denotes memory block value assignment following the τ_e endian format. act (label$_{id}$ | addr) activates an atom specified by the location label or the address value. clratoms removes all the activated atoms from activated atom queue. The remainder three statements are designed for interrupt relevant behaviours. raise irq raises the specific interrupt according to the given interrupt number irq. enable{irq,...} marks the interrupts declared in the set {irq,...} as enabled while disable{irq,...} runs in the other way around.

In several statements above, the exp plays an important role. The exp denotes the xBIL expression which can be used to describe the arithmetics or boolean operations between the register and memory values. The most commonly used operators such as $\wedge$, $\vee$, $+$, $-$ and xor are supported in exp. It is worth mentioning that we use $\texttt{mem(exp,}\tau_{reg}\texttt{,}\tau_e\texttt{)}$ to denote the value of memory value with the type τ_{reg} located at the address given by exp.

B. *xBIL code segment example*

We present an xBIL code example shown in Fig.3 after describing the details of the xBIL language. In Fig.3, on the top of each code block, we list the ARM 9 assembly code. Below the ARM code, the corresponding xBIL code segment is listed.

We use PC, CPSR, IRQ, IF, DEC, EC, FIQ, CS, WC and all the register names to represent the corresponding hardware resources of ARM9 family processor. IRQ is the interrupt handling component. IF, DEC and ec are instruction fetchers, decoder and executor respectively. CS is the control signal bus while WC represents write-back controller. There are six instructions in this example which can be represented by the corresponding atoms of xBIL. Considering the first instruction ADD R1,R1,R2, the first resource zone action represents that instruction fetcher, memory bus and PC register will be occupied for 1 cycle time. During this time, no other atom resource zone can be applied to occupy these hardware resources. Meanwhile, PC is increased and the new atom which the PC points to will be activated. The second zone action occupies instruction decoder and control signal bus for 1 cycle time. The last action performs the calculation and occupies the executing component, R1 and R2 for 1 cycle time.

xBIL brings a variety of benefits to describe the behaviours of the processors, especially the low-level manipulations. With the help of xBIL, it is possible to extract a more accurate program model from one specified executable.

C. Language Extraction

For performing analysis and verification of interrupt relevant properties from the binary code perspective, the approach to extracting the program instructions and interrupt related information should be taken into account by priority. Nonetheless, as of now, there are too many processor architectures. Without involving an intermediate instruction set, we have to choose various techniques to deal with the corresponding processor architectures. IDA Pro [7] is an useful tool to extract the assembly code from executables. Meanwhile, it helps to detect the segments of the executables. These segments comprise the

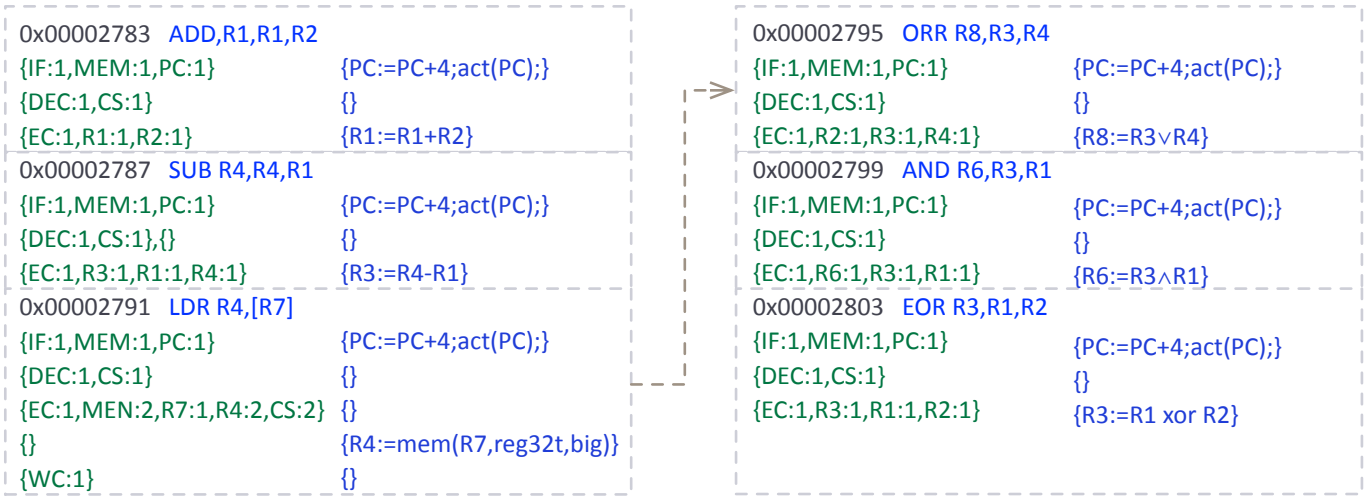

Fig. 3. Sample xBIL code for ARM 9 processor

instructions and the symbol tables. We employ the IDA Pro to obtain the code segments and the corresponding assembly code for one specified hardware platform. Afterwards, the tool xBVM we developed is applied to lift the assembly code to xBIL atoms. The context [1] of xBIL can only be constructed by specifying the processor model and information of the corresponding segments obtained by IDA Pro. The interrupt routines can also be identified by analysing symbol table or initialisation assembly code. For different executable capsulation format, the definitions of the ISRs look definitely different. That is the reason why we have to specify the processor family model when constructing the context of the xBIL code.

IV. Modelling the xBIL Program

This section presents our approach regarding the modelling of binary programs by means of the control-flow graph (CFG) [5], data-flow graph (DFG) [6] and the labelled transition system (LTS). As mentioned previously, xBIL is a binary intermediate language supports multiple architectures. It is extracted from the firmware of real-time embedded system. The extracted xBIL code reserves the interrupt and time relevant features. This brings new challenges to construct the control-flow graph and data-flow graph models. We firstly discuss the modelling of the CFG model. Afterwards, we show how to build the data-flow graph. At the end, the final LTS model which combines the two models mentioned above will be talked about.

A. Building the CFG Model

CFG uses graph notation to capture all the paths that might be traversed in a program during its execution. Distinguished from the source code based CFG construction, our target is rebuilding the CFG based upon the xBIL code. The definition of xBIL program CFG is given below.

1) CFG Model Definition: Let $\mathcal{R}$ be the finite set of registers used in the xBIL program. **ADDR** be the set of all the memory addresses. Let τ_{REG} be the set of all the register types. Let **States** be the set of functions from $\mathcal{R} \cup$ (**ADDR**,τ_{REG}) to $\mathcal{N}$. Intuitively let $s \in$ **States**. For every $r \in \mathcal{R}$, $s(r)$ gives the possible values of the register r in the state s, while for every $addr \in$**ADDR**, $\tau_{reg} \in \tau_{REG}$, $s((addr, \tau_{reg}))$ gets the values of the corresponding value located at address $addr$ in the memory. Let **Stmts** be the set of statements in the atoms. $\tau(addr, \tau_{reg})$ represents the type τ_{reg} of the value which is located at memory address $addr$. $\tau(r, \tau_{reg})$ represents the type τ_{reg} of the value stored in register r.

A control-flow graph (CFG) is a tuple $G_{\mathcal{C}} = (N_{\mathcal{C}}, I_{\mathcal{C}}, E_{\mathcal{C}}, n_{init}, N_{term})$, where $N_{\mathcal{C}}$ is a finite set of nodes corresponding to the control points of the xBIL program, $I_{\mathcal{C}}$ is a finite set of xBIL atoms or interrupt events involved in the program. $E_{\mathcal{C}} : N_{\mathcal{C}} \times I_{\mathcal{C}} \times N_{\mathcal{C}}$ is a finite set of edges, each of them associated with an atom of the xBIL program. We write $n_{\mathcal{C}} \xrightarrow{i} n'_{\mathcal{C}}$ for every $(n_{\mathcal{C}}, i, n'_{\mathcal{C}})$ in $E_{\mathcal{C}}$. n_{init} indicates the initial node of the CFG. N_{term} denotes the set of end nodes. There are another two functions also need to be considered, the first one is $\varrho : N_{\mathcal{C}} \rightarrow$**States** that associates to each node n a possible state of the program at the control point $n_{\mathcal{C}}$. The second one is $\epsilon : I_{\mathcal{C}} \rightarrow$**Stmts** which associates the atom or interrupt event i to the corresponding statements.

We use $\mathcal{O}_{\mathcal{C}} : N_{\mathcal{C}} \times 2^{AP}$ to represent the atomic propositions of the node $n_{\mathcal{C}}$. The construction of function $\mathcal{O}_{\mathcal{C}}$ can be depicted following the rules below:

$$\tau(r, \tau_e) \rightarrow ('\tau(r, \tau_e)' \in \mathcal{O}_{\mathcal{C}}(n_{\mathcal{C}})), \text{ where } \forall r \in \mathcal{R}.$$

$$s(r) = val \rightarrow ('r = val' \in \mathcal{O}_{\mathcal{C}}(n_{\mathcal{C}})), \text{ where } \forall r \in \mathcal{R}, s = \varrho(n_{\mathcal{C}}).$$

$$s((addr, \tau_{reg})) = val \rightarrow ('M_{(addr,\tau_{reg})} = val' \in \mathcal{O}_{\mathcal{C}}(n_{\mathcal{C}})), \text{ where}$$

$\forall addr \in$**ADDR**, $\tau_{reg} \in \tau_{REG}, s = \varrho(n_{\mathcal{C}})$.

Considering the interrupt behaviours of the xBIL program, all the possible branches caused by interrupt must be taken into account as well. In our framework, we use lock area (LA) to assist the construction of the CFG model. The LA marks whether one specific interrupt can be triggered between two nodes. For conveniency, we use **INT** to denote the set of all the interrupt numbers. **LA** indicates the function **LA**: $N_{\mathcal{C}} \rightarrow$ **INT**^{n}.

2) Constructing the CFG Model: Constructing the CFG of a binary program is a complicated process, especially for the binary programs. The difference between the traditional program CFG and xBIL CFG is the elements labelled for each transition. The edge between the control points in traditional CFG is the statement or instruction, while in xBIL, the edge represents the corresponding atom. For xBIL, the basis of whether there is an edge between the control points depends on whether the PC manipulation statement is comprised in the atom corresponding to the transition. In the definition we mentioned above, we have introduced the approach to constructing the basic CFG model. Most of the nodes and edges can be built according to the definition directly. However, this constructed model is still incomplete to comprise all the possible needing branches. The interrupt handling sub-graph and time cost are not taken into consideration in the initiatory CFG. In the remainder part, we explain how to improve the initiatory CFG with ISR sub-graph and label the time cost to the corresponding CFG transition.

Sub Graph Combination Considering the insertion of the interrupt service routines, firstly, we calculate the **LA** of the main program CFG. Meanwhile, all the other CFGs of the ISRs should be prepared for the following insertion actions. The construction of the whole CFG is a recursive iteration process using the algorithm given below.

Algorithm 1 Merge ISR CFG

```
function MERGEISRCFG(CFG g, INT depth)
    LA_g ←calculateLA(g)
    for all node n in g do
        if LA_g(n) ≠ ∅ ∧ n is not visited then
            for all interrupt number j in LA_g(n) do
                g' ← CFG_j
                mark_depth(g',depth+1)
                g←insert(n,g')
                if depth < MAX_DEPTH then
                    mergeISRCFG(g,depth+1)
                end if
            end for
        end if
    end for
end function
```

In Algorithm 1, the initial inputs are the CFG of main program and the nest depth 0. calculateLA(g) calculates the lock area for the specified CFG g. The statements enable{irq,...} and disable{irq,...} take effects on the lock area for the next control point. When we are calculating the lock area, we need to analyse the statements comprised in the atoms simultaneously. CFG_i denotes the CFG corresponding to the ISR of interrupt which number is i. insert(n,g') adds an edge between the initial node of g' and n, this merges two graphs at node n. mark_depth(g',depth+1) add $\delta = n$ (n equals to the second parameter of mark_depth function) proposition to each node in graph g', this proposition helps to indicate the nest depth of the ISR. MAX_DEPTH is the maximum nest depth of the specific hardware platform.

Time Cost Calculation After we merge all the CFGs of ISRs to the main program CFG, the last thing we need to take into consideration is the time cost of the xBIL program. As mentioned previously, the language we proposed is hardware resource oriented, this provides the basis of calculating the more precise execution time of all the instructions. The sample code in Fig. 3 shows the xBIL code of an ARM 9 processor with 5 level pipelines. In our previous work [1], we discussed the execution mechanism of the xBIL code. The actual running time is possible to be analysed considering the pipeline and many other hardware features.

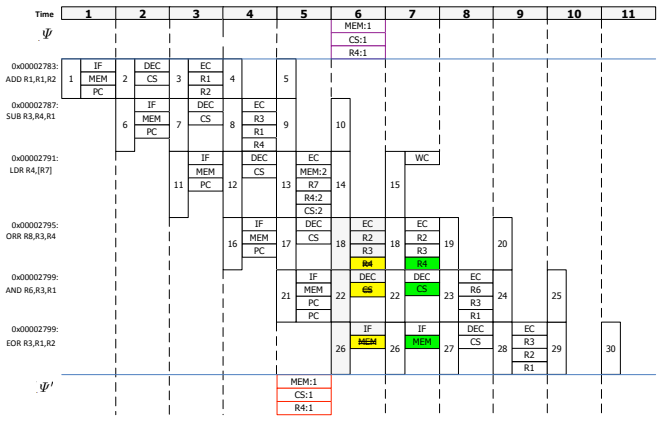

Fig. 4. Example of xBIL time cost calculation

In Fig. 4, we demonstrate the execution time flow of the code segment mentioned in Fig. 3. In the flow table, we can find that after the fifth cycle, the hardware resources MEM, CS and R4 have not been released. Thus, in the following cycle, the next activated atom cannot execute the statements which require the occupied resources mentioned above. In this case, the activated atoms have to be delayed to next cycle. Using xBIL, it is possible to calculate the more precise execution time for the given instructions. The time flow diagram like the Fig. 4 should be built for all the branches of the merged CFG. Afterwards, the corresponding time cost will be noted and finally represented as the labelling function Γ: $I_{\mathcal{C}} \times \mathcal{N}$.

B. Building the DFG Model

DFG is a directed graph that shows the data dependencies among a number of data operations. Formally, we define a DFG as a tuple $(N_\mathcal{D}, E_\mathcal{D}, \mathcal{L}_\mathcal{D})$, where $N_\mathcal{D}$ is the set of nodes and $E_\mathcal{D}$ denotes the set of connections between the input and output ports. Each node in $N_\mathcal{D}$ has its own input/output ports. $\mathcal{L}_\mathcal{D}$ is a labelling function, $\mathcal{L}_\mathcal{D} : N_\mathcal{D} \times (N_\mathcal{C} \cup nil)$ marks the corresponding control point in CFG for each data node. The data node stores the variables or constant values. If one node represents an xBIL atom, it is unnecessary to mark the corresponding atom transitions in the CFG. nil represents that there is no corresponding state node for one specific atom value node. We employ $\mathcal{O}_\mathcal{D} : N_\mathcal{D} \times 2^{AP}$ to compute the atomic propositions for one specific DFG node $n_\mathcal{D}$.

Distinguished from normal DFG, the DFG of xBIL program has its own features. The value of registers or memory may be nondeterministic after performing data operations. For instance, the mem expression which is used to evaluate the content of memory, as we do not know the extract value of the memory in runtime, the new value of the corresponding expression is unknown. This case is also suitable for the expression regarding the registers. In such cases, we employ $\theta((add, \tau_{reg}))$ or $\theta(r)$ to denote the unknown value of the memory or register in runtime.

To avoid redundancy, the DFG contains the registers such as PC and SP only when they are involved in normal calculation, or it will be ignored. At the end, all the calculations in the xBIL code DFG are bit-wised. That may lead to some results of operations are different from the mathematical calculation results (e.g., data overflow), although this matches the reality.

The purpose of analysing the DFG model comes from the requirement of checking the memory access behaviours. For determining the access conflict caused by ISR, we have to clearly know the relations of all the memory access addresses. As our approach is static analysis based, thus without data-flow analysis, it is hard to perform model checking for interrupt relevant properties. Using the static analysis and symbolic execution, we attempt to determine all the possible values of both the memory and registers. For instance, there are two write statements, the first one writes value to add_1, where the second one to $addr_2$. In case we have $addr_1 = addr_2 - 2$, the accessing address of write(val_1,$addr_1$,τ_{r32},τ_s) and write(val_2,$addr_2$,τ_{r16},τ_s) may be overlapped. If we do not know the value of $addr_1$ and $addr_2$, we cannot know the relation between these two addresses either. Thus, this example shows the importance of the analysis of DFG. We can find more potential problems once the values of DFG model are calculated.

C. LTS Combination and Abstraction

The CFG and DFG models illustrate the xBIL program in two perspectives. Analysing these two models separately is helpful to simplify the problem. Nonetheless, for checking the program, the final model should be a comprehensive model including all the desired features. In this part, we introduce how we combine the CFG and DFG models together and abstract a combined model by applying programming slicing technique. The LTS model is similar to the state-event model, where all the events can be treated as intermediate states in this mode.

Given a CFG $G_\mathcal{C} = (N_\mathcal{C}, I_\mathcal{C}, E_\mathcal{C})$ and a set of DFGs, the first DFG in the set is denoted as $G^1_\mathcal{C} = (N_\mathcal{D}, E_\mathcal{D}, \mathcal{L}_\mathcal{D})$, an LTS $(S, Init, P, \mathcal{L}, T)$ is constructed as follows by combining the $G_\mathcal{C}$ and $G^1_\mathcal{D}$. Then, all the other DFGs can be iteratively added in the similar way.

The LTS states S are consisted by the nodes $N_\mathcal{C}$ and event states corresponding to $I_\mathcal{C}$ of CFG. P is the set of atomic propositions of LTS. $\forall i_c \in I_\mathcal{C}, i_c \in P$. $\forall n_c \in N_\mathcal{C}$. If $\alpha \in \mathcal{O}_\mathcal{C}(n_c)$, then $\alpha \in P$. $\forall n_d \in N_\mathcal{D}$. If $\mathcal{L}(n_d) \in N_\mathcal{C}$ and $\alpha \in \mathcal{O}_\mathcal{D}(n_d)$ then $\alpha \in P$. $\mathcal{L}$ is the propositions labelling function where $\forall n_c \in N_\mathcal{C}$, $n_c \to \mathcal{O}_\mathcal{C}(n_c) \in \mathcal{L}$. $\forall i_c \in I_\mathcal{C}, s_{i_c} \to i_c \in \mathcal{L}$, where s_{i_c} is the state corresponding to the event i_c. Meanwhile, $\forall n_d \in N_\mathcal{D}$, if $n_c = \mathcal{L}_\mathcal{D}(n_c)$, then $n_c \to \mathcal{O}_\mathcal{D}(n_d) \in \mathcal{L}$. The transition relations of LTS can be denoted by T. if $n_c \xrightarrow{i} n'_c \in E_\mathcal{C}$, then $s \to s_{i_c} \in T$ and $s_{i_c} \to s' \in T$, where s and s' are the states of corresponding nodes in CFG, s_{i_c} is the state corresponding to the event i_c. Using the rules, the rough LTS can be built in a straightforward way.

Nonetheless, this model contains many states or transitions may not be used in the model checking, or not be referenced by the property formula. The circle paths may lead to the result of time cost calculation infinite even for the fixed step $'for'$ loop. For addressing these problems, we discuss two useful approaches to reducing the states scale and unfold the fixed step loop statically.

1) Post Process: Usually, there are more than 2 million instructions for the embedded system firmware which is designed with complicated logic. Performing verification techniques on raw instruction sequence directly usually cuases state-space explosion. In this scenario, program slicing is a very helpful approach for reducing the model checking state-space. We use an analogous way to reduce the states scale. The brief description regarding this method is listed in the following part. Meanwhile, considering the calculation of time cost of the xBIL program, we demonstrate the solution to unfolding the loops in LTS model as well.

2) Property Based State Reduction: Distinguished from traditional property based slicing technique, we take the references to the operations into account instead of variables. As the variables depend on the context bindings, there is no concrete register referenced by the property formula in some situations. All the concrete registers and addresses may be replaced by variables which is not part of the xBIL program.

In traditional slicing technique, the slicing criterion $C = (x, V)$, where x is a statement in program P and V is a subset of variables in P. Instead of referring to the variables in the program, our solution uses the operation identifiers of the xBIL program. Analogously, the slicing criterion of LTS model is $C_{LTS} = (s, OP)$, where s represnts the state of LTS model and OP is a subset of operations in LTS model. All the events states with operations specified in OP should be reserved. Meanwhile, the dependency-states of these reserved states will also be saved. As the LTS contains both the event states and normal states, once an event state is chosen to be

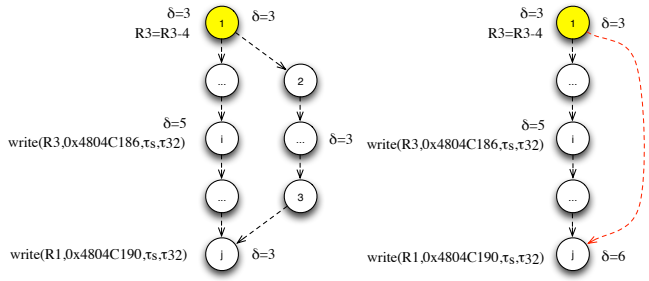

Fig. 5. State Reduction for LTS model

removed, the immediate predecessors and successors will also be removed from the LTS model. One notable thing is that, the time cost marked on the reduced states should be added to new successors of reduced states' predecessor. If there are more than one to the new successor, only the maximum total time cost will be added to the new successor. In practical, we usually choose the terminate state of the program as the first parameter of slicing criterion and all the referenced operations as the second one. The remainder part of the slicing process are performed as normal slicing approach.

The model on the left of Fig. 5 is the original LTS model. We are going to use $C = (j, \{\texttt{write}\})$ to slice the model. Right side model is the reduced model which has removed the event state 7 and the relevant states. The total time cost of state j is updated to 6, due to the state 7 cost 3 time units, and this 3 time units are added to state j.

3) Loop Resolution: In case we need to check the time relevant properties, the time cost calculation is of vital importance for model checking. However, in our model, the circles in LTS may make the result hard to be computed. Even for a fixed step loop, we have known the lower and upper bound of loop number, this information cannot be reflected in LTS model. Unfolding the loop is a straightforward approach to tackling this challenge. Before unfolding the body of the loop, we need to analyse the loop bounds for each branch. We employ the approaches from [9] and [4]. When we detect a circle in LTS, we will attempt to resolve the bounds for each branch which can escape from this circle. In this scenario, the xBIL statements labelled on the transitions of LTS model will be flattened in form of normal procedural program. Each branch which can jump outside the loop will be sliced as a dedicate program. The sliced programs and all the known variable values will be treated as the input to the algorithm mentioned in [4]. If the bounds of all the branches can be resolved, the original LTS circle will be replaced by the unfolded branches directly.

In Fig. 6, we demonstrate a simple example of unfolding the loops of LTS model. The original circle contains two branches which can jump out of the loop. By analysing the bounds of the given branches, we can know the lower and upper bounds of one specific branch. In this example, the lower and upper bounds can be decided using the algorithm provided by [4]. Then we unfold all the possible paths based upon the bounds analysis result. The circle can be unfolded to 11 paths in this

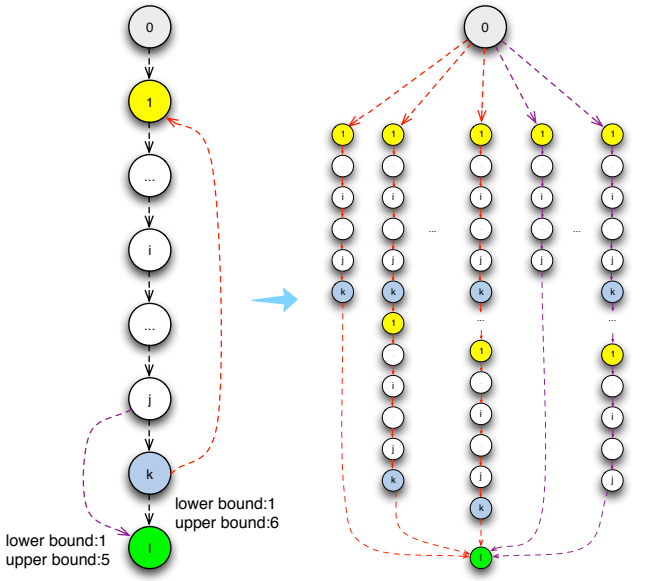

Fig. 6. Unfolding the loops

example. The disadvantage of this approach is the scale of states will speedily grow.

V. Properties and Verification

The challenge of checking the programs with interrupt involved is how to model the relevant properties, on the other hand, how to distinguish the safe and dangerous behaviours is difficult in program's perspective. In our approach, we describe the properties by identifying the relations of LA and the manipulations on specific address. These relations comprise temporal properties, data relations between the accessing addresses. In this section, we firstly introduce the extended logic language RT−CTPL we proposed. Afterwards, model checking of RT−CTPL will be discussed.

A. RT-CTPL

Although the complexity of xBIL is much more simpler than traditional ISA. It is still non-trivial to enumerate all the registers when we apply the CTL [10] or other temporal logic languages. For example, the analysis of security properties have to list all the registers while expressing the movement of a piece of data from memory to a register. The CTL formula should be in following form:

$$EF(mov(R0, data)) \vee EF(mov(R1, data)) \vee$$
$$EF(mov(R2, data)) \vee EF(mov(R3, data)) \ldots$$

Such a long formula is just used to express the value of an existing register is set to the memory located at address data. The CTL formula is quite complicated even for such a simple statement. To address this problem, a logic called CTPL which allows predicated over temporal logic was proposed. CTPL (Computation Tree Predicate Logic) [8] is a new way to describe the temporal predicates of binary programs. The

CTPL can be seen as an extension of CTL with variables and quantifiers which can be in the form $p(x_1, \ldots, x_n)$, where x_i is a free variable or constant. Free variables can get their values from a finite domain. Variables can be universally or existentially quantified. The formula, $\exists r\ EF(mov(r, data))$, expresses the same property compared with the CTL formula above.

1) Syntax of RT−CTPL: However, the CTPL is designed to formulate the properties of malware behaviours of x86 platform. For embedded system, especially real-time embedded system, many features cannot be illustrated by CTPL very well. For addressing this problem, we propose an extended version of RT−CTPL. Compared with CTPL, the new features of RT−CTPL include interrupt operators and the execution time (instruction cycle numbers) operator. The syntax of RT−CTPL is given as follows:

$$\varphi,\psi =_{df} p(t_1,\ldots,t_n) \mid b(u_1,\ldots,u_n) \mid \neg\varphi \mid \mathbf{AX}\varphi \mid \mathbf{AF}\varphi \mid \mathbf{AG}\varphi \mid \mathbf{EX}\varphi \mid \mathbf{EF}\varphi \mid \mathbf{EG}\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \mathbf{E}[\varphi \mathbf{U} \psi] \mid \mathbf{A}[\varphi \mathbf{U} \psi] \mid \exists v.\varphi \mid \forall v.\varphi$$

$p(t_1, \ldots, t_n)$ is a predicate of the arity $n \geq 0$, where $t_1, \ldots t_n$ are terms, $t \in \mathcal{V}$. $\mathcal{V}$ can be $\mathcal{R}$, **ADDR**, $\mathcal{N}$ or variables referenced in sub formula $\exists v.\varphi$ and $\forall v.\varphi$. $b(u_1, \ldots, u_n)$ represents the boolean expression which contains both the time and interrupt operators. Γ^t_{max}, Γ^t_{min}, $\delta^{\to n}$, variables and constant number are the basic elements of $b(u_1, \ldots, u_n)$ connected by arithmetic and boolean operators. No matter what kind of operations are involved in $b(u_1, \ldots, u_n)$, the final expression must be a boolean typed. $\Gamma^t_{max}(\varphi_A)$ and $\Gamma^t_{min}(\varphi_A)$ respectively hold when the maximum or minimum time cost is t for all the paths selected by formula φ_A. In this case, nothing but **AX**, **AF** and **AU** can be used in φ_A. $\delta^{\to n}$ holds when the interrupt nest level of the corresponding state equals n. $b(u_1, \ldots, u_n)$ is the boolean expression construsted by the elements and operations mentioned previously. The truth of $b(u_1, \ldots, u_n)$ will be resolved by SMT solver when we are performing the model checking of the program model. The remainder part of RT−CTPL syntax follows the definition of traditional CTPL [8] formula.

2) Semantics of RT−CTPL: For explaining the semantics of RT−CTPL, we involve a set of preliminary definitions for later use. Firstly, we introduce the concept regarding the context $\mathcal{B}$ which contains pairs of variable names and constant values. $\mathcal{B}$ can be treated as a lookup table for searching all the possible values of one specific variable. The context $\mathcal{B}$ is a partial function maps all the free variables to the possible values. $\mathcal{B}[x \to c]$ indicates the context maps the variable x to c. $\mathcal{B}(t)$ denotes the value mapped to the term t in the context $\mathcal{B}$. $\mathcal{M}$, s $\models_{\mathcal{B}} \varphi$ means the formula φ holds at state s in LTS $\mathcal{M}$ for the given context $\mathcal{B}$.

We list the semantics of the initial and extended part of RT−CTPL in the followig list. The semantics given in this part shows the extended part of the RT−CTPL. Interrupt nest level and time cost calculation operators play the most important role in RT−CTPL. In the following subsection, we will discuss the details regarding the model checking and relevant information. Afterwards, in section VI, we will demonstrate how to use the RT−CTPL to formulate the interrupt relevant properties.

$\mathcal{M}$, s $\models \varphi$	$\Leftrightarrow$	There is a $\mathcal{B}$ that $\mathcal{M}$, s $\models_{\mathcal{B}} \varphi$.
$\mathcal{M}$, s $\models_{\mathcal{B}} \Gamma^t_{max}(\varphi_A)$	$\Leftrightarrow$	$\mathcal{M}$, s $\models \varphi_A$ and $\mathcal{B}[t \to n]$, where n is the maximum total time cost for all the pathes selected by formula φ_A from state s.
$\mathcal{M}$, s $\models_{\mathcal{B}} \Gamma^t_{min}(\varphi_A)$	$\Leftrightarrow$	$\mathcal{M}$, s $\models \varphi_A$ and $\mathcal{B}[t \to n]$, where n is the minimum total time cost for all the pathes selected by formula φ_A from state s.
$\mathcal{M}$, s $\models_{\mathcal{B}} \delta^{\to n}$	$\Leftrightarrow$	holds when $\mathcal{B}[n \to c]$ and c is the interrupt nest level of state s.
$\mathcal{M}$, s $\models_{\mathcal{B}} b(u_1, \ldots, u_n)$	$\Leftrightarrow$	$b(\mathcal{B}(u_1), \ldots, \mathcal{B}(u_n))$ is true.

B. Model Checking The xBIL Program

The model checking algorithm of RT−CTPL is an extension of the algorithm for CTPL. Nonetheless, RT−CTPL allows time cost operators and interrupt nest level identifier, the algorithm has to be able to resolve the RT_{exp} expression mentioned previously. This increases the complexity of the checking algorithm. Another outstanding distinction between the checking algorithm of CTPL and RT−CTPL is the binding or evaluation order of the variables in RT−CTPL formula. The iteration order of CTPL algorithm depends on the size, and the order of RT−CTPL relies on the evaluation dependencies of variables simultaneously.

For presenting the details of the model checking algorithm, we employ $\mathcal{L}_{RT} \subset (S, \Phi, 2^{2^A})$ to represent the relation of states and formulas, where S is the set of states, Φ is the set of all RT−CTPL formulas, and 2^{2^A} is the power set of all possible bindings.

The skeleton of checking algorithm is given as below:

Algorithm 2 Model Checking the RT−CTPL

function MODELCHECKRTCTPL(LTS $\mathcal{M}$, RT−CTPL formula φ)
  for all subformulas φ' of φ in dependency order **do**
    if type of φ' in following list **then**
      $\perp$: label no states
      $p(t_1, \ldots, t_n)$: $LABEL_p(\varphi')$
      $b(u_1, \ldots, u_n)$: $LABEL_b(\varphi')$
      $\neg\varphi$: $LABEL_\neg(\varphi')$
      $\exists v\varphi$: $LABEL_\exists(\varphi')$
      $\varphi \wedge \psi$: $LABEL_\wedge(\varphi')$
      $\varphi \vee \psi$: $LABEL_\vee(\varphi')$
      $\mathbf{E}[\varphi \mathbf{U} \psi]$: $LABEL_{EU}(\varphi')$
      $\mathbf{EX}\varphi$: $LABEL_{EX}(\varphi')$
      $\mathbf{AF}\varphi$: $LABEL_{AF}(\varphi')$
    end if
  end for
  return all $s \in S$ where $\exists C(s, \varphi, C) \in \mathcal{L}_{RT}$
end function

In every iteration over the sub-formulas φ' of φ, all the states in which φ' holds are labelled with φ'. $\mathcal{M} \models \varphi$ holds if the initial state s_0 of $\mathcal{M}$ is finally labelled with φ. For each assignment of all the variables, we can note it as $a.\ a_1 \ldots a_n \in$

A, thus $\mathcal{B} \subset A$. The empty binding which does not comprise any assignment should be noted by $\top$. It formulates an empty condition that is always true. The $C = (\mathcal{B}_1 \vee \cdots \vee \mathcal{B}_n)$ denotes the disjunction of individual bindings. The algorithm 2 labels different kind of φ' by using sub-algorithms. The details of these sub-algorithms can be found in [8]. Compared with the sub-algorithms mentioned in algorithm 2, the first extended part is the iteration which should be performed following the dependency order of the sub-formulas. The second extended part is the labelling function called $LABEL_b$, now we discuss the details of $LABEL_b$ algorithm in Algorithm 3.

Algorithm 3 $LABEL_b$ Algorithm

```
function LABEL_b(φ') //φ' = b(u_1, ..., u_n)
  for all s ∈ S do
    C':= ⊥
    for all φ_A in the Γ terms of φ' do
      if ∃C with (s, φ_A, C) ∈ L_RT then
        C':= C' ∨ C
      end if
    end for
    if C' ≠ ⊥ then
      B:=⊤
      for i:=0 to j do
        v:= time value of Γ_i and γ_i := v
        B:=B ∧ (γ_i = v)
      end for
      for i:=0 to k do
        v:= interrupt nest leval of δ_i and d_i := v
        B:=B ∧ (d_i = v)
      end for
      if b(c_0, ..., c_t, γ_0, ..., γ_j, d_0, ..., d_k) is true and
       B ≠ ⊥ then
        L_RT:= L_RT ∪ (s, φ', {B})
      else
        continue state iteration
      end if
    end if
  end for
end function
```

In Algorithm 3, firstly, we find all the C for each φ_A of Γ formulas. Γ formulas include $\Gamma^t_{max}(\varphi_A)$ and $\Gamma^t_{min}(\varphi_A)$. If we can find the corresponding C, that means every $varphi_A$ is true and has been labelled in $\mathcal{L}_{RT}$. If there is no conflict in the disjunction of all the C, we start to calculate the value of time cost and interrupt nest level identifiers. All the evaluated results are paired with the variables in the corresponding identifiers to form the assignment a mentioned above. Meanwhile, all the assignments are added to context binding $\mathcal{B}$. If $b(c_0, \ldots, c_t, \gamma_0, \ldots, \gamma_j, d_0, \ldots, d_k)$ is true decided by SMT solver, then we add the state s, φ' and binding $\mathcal{B}$ computed previously into the label set $\mathcal{L}_{RT}$. $c_0, \ldots, c_t$ are terms which do not contain time cost and interrupt nest level identifiers, $\gamma_0, \ldots, \gamma_j$ are time cost identifiers and $d_0, \ldots, d_k$ are interrupt nest level identifiers. The terms $c_0, \ldots, c_t, \gamma_0, \ldots, \gamma_j, d_0, \ldots, d_k$ correspond to $u_1, \ldots, u_n$ in formula φ'. Conversely, if the b is not true, we will abandon all the computation results in previous steps and continue the state iteration.

In the next section, we will evaluate the given algorithm above on practical systems to prove the feasibility.

VI. Evaluation

We have evaluated our approach on several systems, including real-time operating system and fuel injection system, etc. In these cases, we illustrate the safety problems and demonstrate the details of the relevant properties which can be applied to describe the corresponding problems. These cases comprise the interrupt properties in multiple perspectives. For each case, we will introduce the background information first. Afterwards, we list the corresponding property formulas. At the end, explanation of the detected problems and summarisation of the case are given.

A. Real-Time Operating System

We have applied our approach to the interrupt safety verification of a real-time operating system - `ORIENTAIS` which is a commercial implementation developed by iSoft Infrastructure Software Co., Ltd. The `ORIENTAIS` is an `OSEK` standard based commercial real-time embedded system for automotive industry. The binary comes from the firmware of one `ECU` built upon this system. By using our approach, several potential interrupt relevant problems caused by memory access conflicts are detected. All these problems are confirmed by the vendor of `ORIENTAIS`. In this part, we will demonstrate how to attack these problems by formulating invalid memory access behaviours caused by interrupts. As the `ORIENTAIS` is an `OSEK` standard based `RTOS`, in this part, we firstly introduce the background information of `ORIENTAIS` and the relevant concepts.

Now, `OSEK` (Offene Systeme und deren Schnittstellen fr die Elektronik in Kraftfahrzeugen) is a standards body that has produced specifications for an embedded operating system, a communications stack, and a network management protocol for automotive embedded systems [11]. It has also produced other related specifications. `OSEK` was designed for providing a standard software architecture for the various electronic control units (`ECUs`) throughout a car. It is the most widely used automotive electronic oriented operating system standard all over the world. The target operating system `ORIENTAIS` implements all the features of `OSEK` standard. The firmware we used in this case is compiled for Freescale `9S12` chip.

For detecting the memory access conflicts, the formula given below is applied to check the `ORIENTAIS` firmware. The verification result shows there are 29 analogous problems in the given firmware, which means the memory access conflict problem may be occurred in 29 places of this system.

$$
\begin{aligned}
&\exists addr_1, addr_2, \tau_{reg1}, \tau_{reg2}, \tau_e, val_1, val_2, n_1, n_2 \\
&\quad \mathbf{EF}(\texttt{write}(val_1, addr_1, \tau_e, \tau_{reg1}) \wedge \delta^{\rightarrow n_1} \\
&\quad\quad \wedge \mathbf{EF}(\mathcal{M}_{(addr_2, \tau_{reg2})} = val_2 \wedge \delta^{\rightarrow n_2} \wedge n_2 < n_1) \\
&\quad\quad \wedge \texttt{overlap}(addr_1, addr_2, \tau_{reg1}, \tau_{reg2}))
\end{aligned}
$$

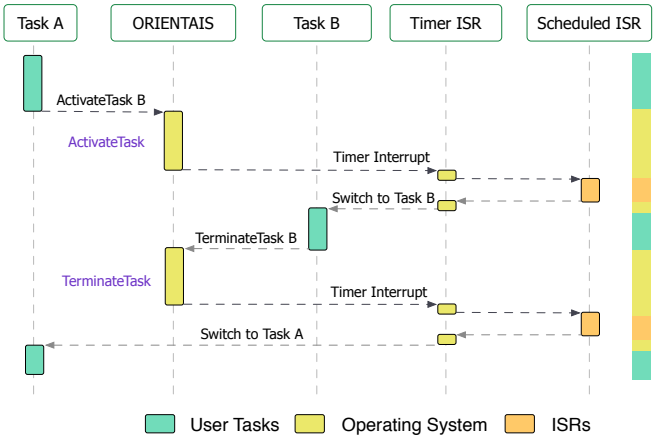

Fig. 7. The interrupt Mechanism in ORIENTAIS

In the formula given above, the $\mathcal{M}_{(addr_2,\tau_{reg2})} = val_2$ and $\texttt{write}(val_1, addr_1, \tau_e, \tau_{reg1})$ represent the AP marked on the state and transition of LTS. $\delta^{\rightarrow n_1}$, $\delta^{\rightarrow n_2}$ and $n_2 < n_1$ denote the interrupt nest depth of the corresponding transition or state are n_1 and n_2, where $n_2 < n_1$. This means the write manipulation is performed in the higher nest depth compared with the memory value evaluation operation. $\texttt{overlap}(addr_1, addr_2, \tau_{reg1}, \tau_{reg2})$ indicates the operation addresses $addr_1$ and $addr_2$ are overlapped. In other words, the read and write operations are performed in the overlapped memory area.

For explaining the details of the problem we found by the formula above, we show an interrupt call flow of this problem in Fig.7. This is a common case of task scheduling. `Task A` uses the OSEK API `ActivateTask` to activate `task B`, the ORIENTAIS switches to a software context switching interrupt ISR. After context switching, `Task B` is activated and in running state. After execution, `Task B` uses OSEK API `TerminateTask` to terminate itself. ORIENTAIS raises the software interrupt again for switching the context to `Task A`. In this case, the scheduled ISR updates the global variable in API `ActivateTask`, the memory access conflict may be occurred. As the variable is not well protected in the corresponding API, the relevant APIs regarding this conflict variable may be failed to complete the task.

Generally speaking, in the implementation of ORIENTAIS, global variables which reflect the states of operating systems are shared in all the APIs for performance consideration. However, the implementation of this optimisation brings potential security risks obviously.

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

B. Fuel Injection System

Fuel injection is a system for admitting fuel into an internal combustion engine. It has become the primary fuel delivery system used in automotive engines, having replaced carburetors during the 1980s and 1990s [12]. A variety of injection systems have existed since the earliest usage of the internal combustion engine. For controlling the engine, usually an EMS (Engine Management System) is embedded as a dedicate ECU (Electronic Control Unit) in the car. When we attempt to start the engine, we send the start engine signal to the EMS (ECU) and wait for the engine to start.

In this case, the Fuel Injection System (abbreviated to FIS) requires injecting the fuel regularly. Each interval between the two contiguous fuel injection actions must be very approximative. In case the interrupts are triggered during the interval of two fuel injection actions and the time cost of the handling program changes timely, the engine may not be started normally or it may bring chronic damages to the engine system.

For guaranteeing the requirement mentioned above, we employ the following formula to describe the corresponding interrupt property:

$$
\begin{aligned}
&\exists \tau_{reg}, \tau_e, val, t \\
&\quad (\mathbf{EF}(\texttt{write(0x00000001,0x4804C190,}\tau_e, \tau_{reg}) \\
&\qquad \wedge \Gamma^t_{max}(\mathbf{AX}(\neg \texttt{write(0x00000001,0x4804C190,}\tau_e, \tau_{reg}) \\
&\qquad\qquad \mathbf{AU} \\
&\qquad\qquad \texttt{write(0x00000001,0x4804C18C,}\tau_e, \tau_{reg}))) \\
&\qquad \wedge t < max\_interval) \\
&\quad \mathbf{EU} \\
&\quad \texttt{write(0x00000001,0x4804EA3C,}\tau_e, \tau_{reg})
\end{aligned}
$$

The EMS system uses GPIO to send signals to the injection component. In this experiment, the developer uses $\texttt{write(0x00000001,0x4804C190,}\tau_e, \tau_{reg})$ to send stop injection signal, where the start injection signal is sent by using $\texttt{write(0x00000001,0x4804C18C,}\tau_e, \tau_{reg})$. `0x4804C18C` is the address of the corresponding GPIO port. According to the definition of RT-CTPL, Γ^t_{max} denotes the temporal formula inside is satisfied and the maximum time cost of all the paths which match the formula is t. The formula $t < max\_interval$ restrains the time cost within $max\_interval$ for either the main program or ISRs. At the end, `0x4804EA3C` is the port address for notifying the start engine process has been finished, no matter what the result is.

We applied our method on one EMS firmware provided by `Changan Automotive`. We have known there is a problem in this firmware, however, even the developers cannot determine what causes this problem. Fortunately, the verification counter example helps us to locate the source. The developer forgets to disable the interrupt triggered by the injection sensor, as a result, the interrupt is triggered many times. This makes the interval between two injections keep changing.

In Fig.8, the chart on the top shows the injection volume and the corresponding time. The x axis is the time unit, and the y is the injection volume percentage. The injection illustrated on the top is irregular and may bring damage to the engine. Compared with the irregular injection chart, another one on the bottom presents the injection intervals of the fixed firmware. By disabling the particular interrupt corresponding to the sampling ISR, the ISR will not be interrupted any

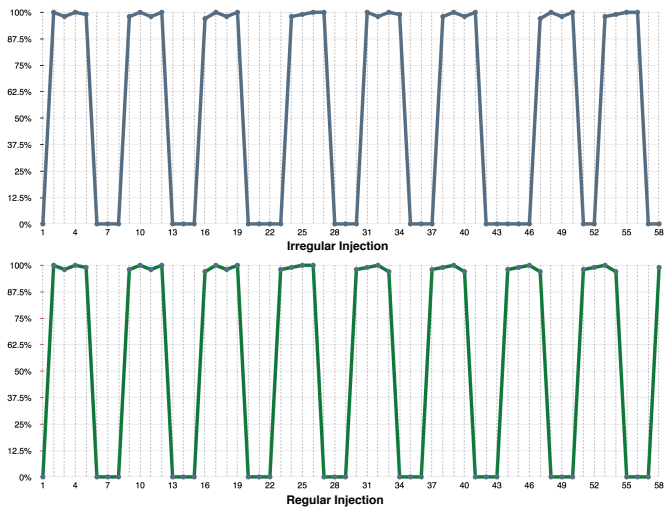

Fig. 8. Injection Interval Comparison

more. The interval between two injections is now regular.

VII. Related Works

The research of interrupts therefore becomes the focus of attention in both industry and academia.

There have been proposals suggesting that interrupts can be regarded as threads and may be verified like threads using some similar verification methods. For example, Hills [13], Regehra and Cooprider [14], Feng et al. [15], the `Nemesis OS`, `TimeSys Linux` [16], and `Solaris` [17] all take the interrupts-as-threads approach.

Some researchers have attempted to apply different formal methods to the interrupts. Palsberg et al. [18–21] have performed a series of studies on interrupt-driven `Z86-based software`. They have proposed a typed interrupt calculus which guarantees stack boundedness and enables modular type checking. And they have also developed a tool to analyse interrupt latencies, stack sizes, deadline, and verify fundamental safety and liveness properties.

Bérard et al. [22–24] have presented a class of interrupt timed automata (`ITA`), which is suited to the description of timed multi-task systems with interruptions in a single processor environment. They have analysed the reachability problem and investigated the verification of some real-time properties over `ITA`. Li et al. [25] have proposed a controller automata as an extension of time automata to model the time behaviours of real-time systems with the nested interrupt handling. They have presented a methodology to analyse the schedulability and some properties of real-time systems with the nested interrupts by composing with an environmental simulation. Nonetheless, the execution time is not accuracy enough. Meanwhile, the lack of liveness properties is another disadvantage of the methods mentioned above.

More related works are presented in [26–28], where kinds of interrupt mechanisms are added to process algebras. For example, Baeten et al. [26] have developed a mechanism, including priorities and non-deterministic choice, in the algebra of communicating processes, which can be used to describe the working of interrupts. It takes a point that interrupt triggered by event, and describes the programs with interrupts by algebra laws. However, it does not involve data states or time. In Communicating Sequential Processes (`CSP`) [29], Hoare has defined a simple form of interrupt $P \triangle Q$. It behaves as P until interrupted by the first event of Q, and after that P is never resumed. Hoare has explained this interrupt by the trace model.

Our work is also related to model checking based malware detection. In this domain, many techniques have been proposed to identify malicious or insecure behaviours [30–36]. Most of them [30, 32] use finite-state automata to model the system, hence these approaches are hard to describe interrupt behaviour. Meanwhile, most of these approaches are designed for the specific hardware platform.

VIII. Conclusion and Future Work

In this paper, we proposed an approach to verifying the interrupt relevant properties of real-time embedded systems from the binary code perspective. Following our approach, the firmware of the embedded system is lifted to an intermediate language that we have proposed. This intermediate language is designed to describe the instructions over multiple hardware platforms. Meanwhile, the `xBIL` intermediate language is used to construct the `CFG` and `DFG` for modelling the checking program. After applying a variety of techniques for combining, abstracting and optimising these models, a simpler `LTS` model is built for performing the model checking. Extending from the `CTPL`, we proposed `RT-CTPL` with several new features relevant with time cost and interrupt behaviours. We then overviewed the model checking algorithm and the corresponding tools. For proving the efficiency of our approach, we applied it to several embedded systems and presented some evaluation cases in this paper. The result is encouraging. The verified systems now have been widely deployed in automotive industry in China. It is worth mentioning that the `OSEK` standard group has certified the `ORIENTAIS` operating system which is verified by the method in this paper.

In the future, we plan to apply this approach to more industry areas, especially the critical embedded systems. The tools should be optimised for checking the large scale models. A cache emulator and more CPU models will be added to our tools for calculating more precise execution time and supporting more hardware architectures as well.

References

[1] 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.

[2] 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.

[3] D. R. Butenhof, *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[4] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution," in *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*. IEEE, 2006, pp. 57–66.

[5] F. E. Allen, "Control flow analysis," in *ACM Sigplan Notices*, vol. 5, no. 7. ACM, 1970, pp. 1–19.

[6] K. M. Kavi, B. P. Buckles, and U. N. Bhat, "A formal definition of data flow graph models," *Computers, IEEE Transactions on*, vol. 100, no. 11, pp. 940–948, 1986.

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

[8] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2005, pp. 174–187.

[9] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper, "Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis." in *WCET*, 2007.

[10] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.

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

[12] H. Fujisawa, T. Iwanaga, and M. Miyaki, "Fuel injection system," Oct. 18 1988, uS Patent 4,777,921.

[13] T. Hills, "Structured interrupts," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 1, pp. 51–68, 1993.

[14] J. Regehr and N. Cooprider, "Interrupt verification via thread verification," *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 9, pp. 139–150, 2007.

[15] X. Feng, Z. Shao, Y. Dong, and Y. Guo, "Certifying low-level programs with hardware interrupts and preemptive threads," in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 170–182.

[16] C. TimeSys, "Timesys linux," *TimeSys Corp. See http://www.timesys.com*, 2004.

[17] S. Kleiman and J. Eykholt, "Interrupts as threads," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 2, pp. 21–26, 1995.

[18] D. Brylow, N. Damgaard, and J. Palsberg, "Static checking of interrupt-driven software," in *Proceedings of the 23rd international conference on software engineering*. IEEE Computer Society, 2001, pp. 47–56.

[19] J. Palsberg and D. Ma, "A typed interrupt calculus," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer, 2002, pp. 291–310.

[20] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T. A. Henzinger, and J. Palsberg, "Stack size analysis for interrupt-driven programs," in *Static Analysis*. Springer, 2003, pp. 109–126.

[21] D. Brylow and J. Palsberg, "Deadline analysis of interrupt-driven software," in *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5. ACM, 2003, pp. 198–207.

[22] B. Bérard and S. Haddad, "Interrupt timed automata," in *Foundations of Software Science and Computational Structures*. Springer, 2009, pp. 197–211.

[23] B. Bérard, S. Haddad, and M. Sassolas, "Real time properties for interrupt timed automata," in *Temporal Representation and Reasoning (TIME), 2010 17th International Symposium on*. IEEE, 2010, pp. 69–76.

[24] B. Berard, S. Haddad, and M. Sassolas, "Interrupt timed automata: verification and expressiveness," *Formal Methods in System Design*, vol. 40, no. 1, pp. 41–87, 2012.

[25] G. Li, S. Yuen, and M. Adachi, "Environmental simulation of real-time systems with nested interrupts," in *Theoretical Aspects of Software Engineering, 2009. TASE 2009. Third IEEE International Symposium on*. IEEE, 2009, pp. 21–28.

[26] J. C. Baeten, J. A. Bergstra, and J. W. Klop, *Syntax and defining equations for an interrupt mechanism in process algebra*. Department of Computer Science, Centrum voor Wiskunde en Informatica, 1985.

[27] B. Diertens, "new features in psf i-interrupts, disrupts, and priorities," *report P9417, Programming Research Group-University of Amsterdam*, 1994.

[28] A. Engels and T. Cobben, "Interrupt and disrupt in msc: Possibilities and problems," in *Proceedings fo the 1st Workshop of the SDL Forum Society on SDL and MSC*, no. 104. Citeseer, 1998.

[29] C. A. R. Hoare, *Communicating sequential processes*. Prentice-hall Englewood Cliffs, 1985, vol. 178.

[30] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2005, pp. 174–187.

[31] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi, "Static detection of malicious code in executable programs," *Int. J. of Req. Eng*, vol. 2001, pp. 184–189, 2001.

[32] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Security and Privacy, 2005 IEEE Symposium on*. IEEE, 2005, pp. 32–46.

[33] A. Lakhotia, D. R. Boccardo, A. Singh, and A. Manacero Jr, "Context-sensitive analysis of obfuscated x86 executables," in *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*. ACM, 2010, pp. 131–140.

[34] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Proactive detection of computer worms using model checking," *Dependable and Secure Computing, IEEE Transactions on*, vol. 7, no. 4, pp. 424–438, 2010.

[35] F. Song and T. Touili, "Pushdown model checking for malware detection," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2012, pp. 110–125.

[36] F. Song and T. Touili, "Efficient malware detection using model-checking," in *FM 2012: Formal Methods*. Springer, 2012, pp. 418–433.