Test Generation for Embedded Executables via Concolic Execution in a Real Environment

ARTICLE *in* IEEE TRANSACTIONS ON RELIABILITY · MARCH 2015

Impact Factor: 1.93 · DOI: 10.1109/TR.2014.2363153

6 AUTHORS, INCLUDING:

Ting Chen

University of Electronic Science and Techn…

24 PUBLICATIONS **48** CITATIONS

Test Generation for Embedded Executables via Concolic Execution in a Real Environment

Ting Chen, Xiao-Song Zhang, Xiao-Li Ji, Cong Zhu, Yang Bai, and Yue Wu

***Abstract*—Traditional software testing methods are not effective for testing embedded software thoroughly due to the fact that generating effective test inputs to cover all code is extremely difficult. In this work, we propose an automatic method to generate test inputs for embedded executables which is based on concolic execution. The core idea of our method is to divide concolic execution into symbolic execution on hosts, and concrete execution on targets, so considerable development work can be saved. Our method overcomes the limitations of the software and hardware abilities of embedded systems by restricting heavy-weight work on resourceful hosts. One feature of our method is that it targets executables, so the source of tested software is not needed. Another feature is that tested programs run in a real environment rather than in a simulator, so accurate run-time information can be acquired. Symbolic execution and concrete execution are coordinated by cross-debugging functions. Then we implement our method on Wind River VxWorks. Experiments show that our method achieves high code coverage with acceptable speed.**

***Index Terms*—Embedded software, concolic execution, cross debugging.**

Acronyms and Abbreviations

API	Application Programming Interface
BAP	Basic Analysis Platform
CEC	Concrete Executor Client
DBA	Dynamic Bitvector Automata
DBI	Dynamic Binary Instrumentator
GAL	Generic Assembly Language
IDE	Integrated Development Environment
IR	Intermediate Representation
S2E	Selective Symbolic Execution
ROM	Read-Only Memory
RPC	Remote Procedure Call
SES	Symbolic Executor Server
WTX	Wind River Tool Exchange
SE	Symbolic Execution
CE	Concrete Execution
cmd	command
pc	path condition
I-O	Input-Output
SC-UE	Strictly Consistent Unit-level Execution
SC-SE	Strictly Consistent System-level Execution

Manuscript received September 22, 2013; revised March 04, 2014 and June 17, 2014; accepted July 11, 2014. Date of publication October 21, 2014; date of current version February 27, 2015. This work was supported by the Fundamental Research Funds for the Central Universities E022050205, Application Research Foundation of Sichuan Province of China 2014JY0168, Software Definition Manufacture Project of National Tech SMEs Innovation Funds 14C26213201050. Associate Editor: C. Smidts.

T. Chen, X.-S. Zhang, X.-L. Ji, C. Zhu, and Y. Wu are with the School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China (e-mail: chenting19870201@163.com).

Y. Bai is with the No. 30 Research Institute of China Electronics Technology Group Corporation (CETC), Chengdu 610093, China (e-mail: 602760688@qq.com).

Digital Object Identifier 10.1109/TR.2014.2363153

I. Introduction

DUE to the pervasive usage of devices with embedded software, software quality is extremely critical to the economy, and to human lives. Testing is a common way to improve software quality. However, traditional methods are not able to test embedded software thoroughly due to the fact that test inputs are not effective enough to cover all possible execution scenarios. As a consequence, we resort to a promising technique, *concolic execution*, which is proven effective at testing software on general-purpose computers.

Symbolic execution, which was first proposed in the 1970s [1], [2], becomes a research hotspot for the significant advancement of software and hardware abilities. Concolic execution [3], also known as dynamic symbolic execution [4] or white-box fuzzing [5], is a variant of symbolic execution which combines symbolic execution with concrete execution.

In plain words, concolic execution consists of running the program under test both concretely, executing the actual program, and symbolically, gathering constraints in terms of input parameters [6]. Accuracy is a remarkable advantage of concolic execution because run-time information is always available. Besides, concolic execution can test programs thoroughly in theory because symbolic execution steers new executions towards unexplored paths.

For its advantages, in recent years, concolic execution has been applied to several research fields such as automatic test generation [3]–[5], [7], vulnerability signature generation [8]–[12], automatic reverse-engineering [13]–[15], and others. However, applying concolic execution to embedded software testing is rather challenging. One obvious reason lies in that the computational ability (especially CPU performance and memory capacity) of embedded systems can hardly meet the rigid requirements of symbolic execution. Besides, existing methods have to port code because some necessary software testing modules for concolic execution, for example instrumentation tools [16]–[18] and theorem provers [19]–[21], are incompatible with embedded systems. But the effort required for code porting can be alleviated to a great extent by our method. Find details in Section II.

In this paper, we propose a different concolic-execution-based approach to automatically generate test inputs for embedded executables in a real environment. The core idea is to divide concolic execution into symbolic execution on hosts, and concrete execution on targets. Therefore, we considerably reduce the manual work for code porting, such as the modification of theorem provers. The intuition behind our method is that heavy-weight symbolic execution has to be confined to resourceful hosts (personal computers, workstations etc.), and light-weight concrete execution should be run on meager targets (embedded systems) to get run-time information.

The coordination of symbolic execution and concrete execution is achieved by cross debugging provided by the vendors of embedded systems. Fortunately, embedded software vendors will provide cross-debugging functions along with corresponding development kits, if they need a third party to develop applications for their embedded systems.

We implement our method on Wind River VxWorks which provides the Wind River Tool Exchange (WTX) protocol for cross-debugging. The implementation of our prototype is also a focus of this work. Then practical case studies verify the viability of our method. Afterwards, the effectiveness and efficiency of our prototype are studied through experiments with benchmark programs. Results report that our prototype achieves high code coverage with acceptable speed.

Contributions of this work are summarized as follows.

1) We apply concolic execution to embedded software by dividing it into symbolic execution on hosts, and concrete execution on targets. As a consequence, accurate run-time information can be acquired. Simulators can hardly operate the same as real devices, leading to inaccurate execution of simulation-based methods. For example, a typical iPhone simulator does not model the accelerometer, the location routines, or the multi-touch interface. Another benefit of our design is that the bottleneck of the software and hardware abilities of embedded systems can be circumvented by restricting heavy-weight tasks on resourceful hosts. Furthermore, by constraining symbolic execution on hosts, substantial manual work for code porting can be saved.
2) As far as we know, we are the first to implement concolic execution on VxWorks.

This paper is organized as follows. Section II reviews related works briefly. Motivations of this work are presented in Section III. Section IV presents the architecture of our method, while Section V describes the algorithm of our method. Section VI details our implementation for VxWorks, then practical cases are studied in Section VII. Section VIII focuses on the experiments and analysis of benchmark programs. Section IX discusses the limitations of our method, as well as possible solutions; and Section X concludes.

II. Related Work

So far, only a few methods have been proposed to apply concolic execution to embedded software. Kim and his colleagues ported CREST [22] and KLEE [23] to some benchmark programs [24]. Similarly, they also applied concolic execution to a mobile device [25]–[27], and a flash memory [28], [29] manufactured by Samsung corporation. The testing processes of the above four tools are similar: 1) instrumenting the source of tested software by the modified CREST or KLEE; 2) concolically executing the instrumented software under test on target systems; 3) gathering path conditions, and generating new test inputs through the modified theorem provers; and 4) repeating the testing progress until any terminal conditions are met. We observe several common features of those methods: 1) the C or C++ source code of the tested software is needed because the methods proposed by Kim *et al.* [24]–[29] are built on top of CREST or KLEE; 2) significant manual work for code porting is required, for example the modifications of CREST or KLEE, as well as theorem provers; and 3) symbolic execution and constraint solving run on the target systems, leading to significant overhead.

Bardin and Herrmann [30], [31] proposed to translate embedded executables into Intermediate Representations (IRs), and then execute IRs concolically in a simulator. Now, Osmose [30], [31] is being redesigned based on the BINCOA framework [32] which provides Dynamic Bitvector Automata (DBA) that is better than its original Generic Assembly Language (GAL). Their methods possess the following features. 1) Specific simulators should be developed according to specific embedded systems. 2) The run-time information obtained in simulators should not be expected to be as accurate because existing simulators cannot behave exactly the same as real devices. As a result, the bugs discovered by Osmose cannot be guaranteed to be real [30], and must be verified.

The proposed method in this work tests the executables of tested programs, so we compare our method with other typical binary-level concolic execution tools hereinafter.

SAGE [5] is a proprietary tool used by the Microsoft Corporation which directly executes machine instructions symbolically without translation to any IRs. Unsurprisingly, SAGE can cope with the x86 architecture only because its symbolic execution is architecture-dependent. SAGE collects concrete execution traces using the iDNA framework, and then symbolically executes those traces based on the TruScan replay framework. The design of SAGE helps to port concolic execution to embedded systems because SAGE can separate symbolic execution from concrete execution easily. However, the portability of SAGE depends on the portability of iDNA and TruScan. So far, we have not seen any reports about the applications of SAGE on embedded systems.

SMAFE was proposed and developed by us, and published in our previous work [33]. Like SAGE, it directly executes machine code concolically. So it is also architecture-dependent, and now it shares the same shortcoming with SAGE that it is able to deal with the x86 platform only. Differently, SMAFE uses Pin[1] to instrument tested programs, so it cannot test embedded applications directly.

More concolic execution tools analyze IRs which are translated from machine code. The advantages are at least two-fold: 1) IR-based concolic execution tools could be architecture-independent, i.e., they can test programs of different platforms, if translation modules for specific architectures are available; and

[1]Pin is a dynamic binary instrumentation framework that enables dynamic analysis.

2) considerable development effort can be saved because IRs are always much simpler than machine code.

Fuzzgrind [7] and Catchconv [34] are two typical IR-based concolic execution tools which use Valgrind to instrument tested programs. So the two tools cannot handle embedded applications directly. Selective Symbolic Execution (S2E) [35] works in a different way: it runs tested programs in QEMU (an open-source virtual machine), then translates machine code into the IR proposed by LLVM (a compiler infrastructure), and finally symbolically executes IRs by KLEE [23]. Therefore, the portability of S2E to specific embedded systems relies on whether QEMU supports that system. Bitblaze [36] collects concrete execution traces of tested programs in TEMU (a modified QEMU), and then symbolically executes Vine (a static analysis component) IRs which are translated from the traces by Rudder (concolic execution component). So Bitblaze shares the same shortcoming in portability with S2E.

As far as we know, MAYHEM [37] is the closest related work to our research. It leverages the BAP framework [38] to translate x86 instructions to IR statements, so MAYHEM is architecture-independent in principle. At a high level, MAYHEM consists of two processes: a Concrete Executor Client (CEC) which concretely executes tested programs on a target system, and a Symbolic Executor Server (SES) which symbolically analyzes tainted blocks on any platforms. The proposed architecture of MAYHEM helps to port itself to any embedded systems because it separates concrete execution from symbolic execution. However, our proposed method differs from MAYHEM in many aspects.

First, the CEC component consists of a taint tracker, a Dynamic Binary Instrumentator (DBI), and a virtualization layer to perform dynamic taint analysis. On the contrary, the target part of our proposed architecture is quite simple as it just has two modules (details will be presented in Section IV): a CE core for concrete execution, and a debug agent for cross debugging. So our method could suit embedded systems better due to their limited hardware resources. Second, researchers need to make non-negligible effort to port the taint tracker, the DBI (based on Pin), the virtualization layer to the target for dynamic taint analysis. Obviously, the porting work is not required for our method.

Moreover, MAYHEM applies hybrid symbolic execution while our method uses concolic execution. So the interactions between concrete execution with symbolic execution show obvious differences. Specifically, MAYHEM runs a tested program concretely on a target, then if any instructions are tainted, the CEC sends them to the SES to perform hybrid symbolic execution (off-line symbolic execution + on-line symbolic execution). Differently, for our method, the SE core executes a tested program symbolically, and queries concrete values of registers, memory locations etc. if needed. From the aspect of implementation, MAYHEM implements a cross-platform, light-weight RPC protocol to connect the CEC and the SES, while our method takes advantage of existing debugging interfaces. So our method is cheaper in development, and it has better portability to embedded platforms.

Finally, Cha *et al.* do not explore the potential portability of MAYHEM. In other words, their work does not present any attempts to apply MAYHEM to test embedded applications, especially in implementations and experiments, even if it is possible in design. Contrarily, our work proposes a general method for embedded applications and details in implementations and experiments on a specific embedded system called VxWorks.

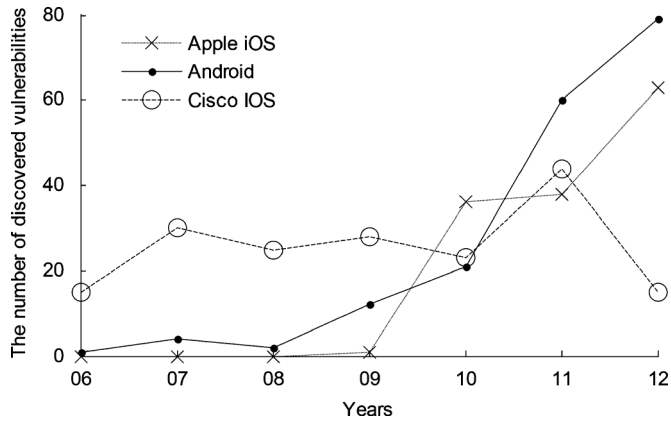

Fig. 1. Trend of the numbers of discovered vulnerabilities from Jan. 1, 2006 through Apr. 26, 2012 (data collected from National Vulnerability Database. http://nvd.nist.gov/).

III. Motivation

The primary reason for us to conduct this work is the **existence of defective embedded software**. Fig. 1 shows the trend of the numbers of discovered vulnerabilities in three popular embedded systems: Android, Apple iOS, and Cisco IOS from January 1, 2006 to April 26, 2012. From the statistics, we find that a number of vulnerabilities are discovered every year even though great effort has been made in software testing.

From our perspective, one major reason for so many vulnerabilities is likely be inadequate testing techniques. Black box testing is the mainstream method to test embedded software. Test inputs are generated randomly [39], [40], or through use cases [41], or through other heuristics [42]. Then test inputs are fed to programs under test, and afterwards the outputs of tested programs are observed and analyzed. If any output is not correct, a potential vulnerability is discovered. Ignorance of the internal states of tested programs is the main drawback of black box testing. As a consequence, black box testing is very unlikely to achieve high code coverage, i.e., a significant number of vulnerabilities may remain in tested programs even if all test inputs generated by black box testing are executed.

Static analysis [43] always acts as a complement of black box testing. More specifically, it analyzes the source code or binaries or any other representations of embedded software without actually running tested programs. Static analysis usually generates massive false alarms due to the loss of run-time information.

The effectiveness of concolic execution, which has been validated by the experiences with general-purpose computers, is our second motivation. A number of concolic execution tools have been developed, including CUTE [3], DART [4], SAGE [5], Fuzzgrind [7], as well as our own SMAFE [33] and SEVE [44] etc, so we have many examples to learn from.

The fact that **instrumentation can be substituted with debugging** so as to obtain run-time information of tested programs strengthens our confidence in porting concolic execution to embedded software. This viewpoint will be validated in the following sections. Instrumentation is a commonly-accepted method in the research field of concolic execution on

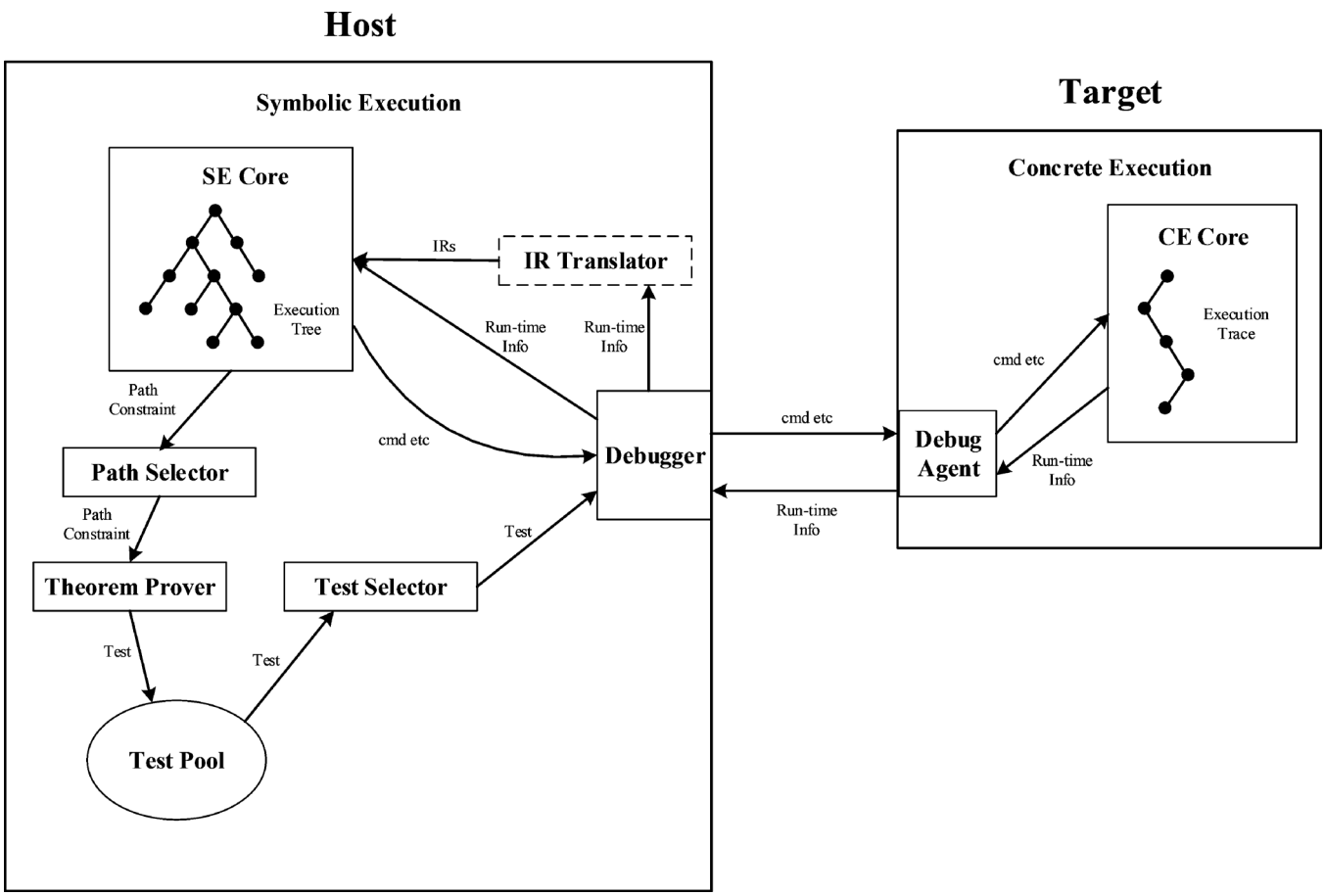

Fig. 2. High-level architecture of our method.

general-purpose computers. However, it could be a problem for embedded platforms.

First, excessive consumption of computing resources makes instrumentation a bottleneck of concolic execution. The performance problem should be more severe for embedded systems due to the meager software and hardware abilities. Second, porting instrumentation tools to embedded systems is a very demanding manual process because instrumentation is highly relevant to operating systems and hardware architectures.

Fortunately, cross debugging, the necessary technique for our method, is always available from the vendors who permit a third party to develop applications for their embedded systems. For instance, embedded Linux can be debugged by GDB; the WTX protocol is the debugging technique provided by Wind River for VxWorks; the debugging function for Windows CE is integrated in the Platform Builder IDE; CodeWarrior Development Studio provides a debugging function for Palm; and, we can get run-time information for Android, and iOS applications through Android Debug Bridge, and debugserver respectively. Thanks to commonly-used cross-debugging techniques, the application of our method to different embedded systems becomes possible.

IV. Architecture

In this section, we first introduce the overview architecture of our method by describing each module, and then state the cooperation of all modules through a typical run of concolic execution. Fig. 2 demonstrates the architecture. Note that only the IR translator is surrounded by a dashed box because it is an optional module.

A. Description of Modules

The Symbolic Execution (SE) core plays as the heart of the symbolic execution part, which maintains the execution tree. The main functions of the SE core include symbolizing inputs, tracking symbols, and generating path conditions. The path selector determines how to modify the path conditions produced by the SE core, and sends the modified path conditions to a theorem prover as its inputs. The design of the path selection algorithm which significantly influences the efficiency of symbolic execution is the key of the module. Popular algorithms include depth-first [3], [4], [34], [45]–[47], breadth-first [36], generational [5], [7], [23], [45], random [23], and meta-strategy [48].

The function of the theorem prover is to judge whether the inputs (i.e., path conditions) are satisfiable. An unsatisfiable judgment of the input set means the tested program cannot exercise the path predicated by the path condition. On the contrary, if the judgment is a satisfiable input set, then an assignment to each input variable is provided. Then the assignment is interpreted as a test input, which is able to drive the tested program to the exact path predicated by the path condition. All test inputs are stored in the test pool.

The test selector chooses a test input from the test pool for the next concolic execution, so it depends on the path selection algorithm. For example, given the depth-first algorithm, the

test selector must choose the test input which will exercise the longest path in the execution tree; if a generational algorithm is employed, the test selector must choose the test input, which will execute the most uncovered code.

The debugger is the unique interface in a host to communicate with a target. The debugger sends run-time information of tested programs to the SE core or the IR translator, which is received from the debug agent. Besides, the debugger receives commands (abbreviated as cmd in Fig. 2), and tested programs from the SE core, and the test selector respectively; and then sends them to the debug agent.

The IR translator is used to translate machine instructions to IR statements. The input for the IR translator is machine code, and the output is IRs which can be interpreted by the SE core. The IR translator depends on the target architecture and the SE core, so it is an optional module (dashed box) in our method. For example, our implementation ports SMAFE for x86-Windows to x86-VxWorks, so we do not import a translation module. In other situations, the IR translator is necessary if we want to port an IR-based concolic execution tool to embedded systems (see details in Section VI).

The structure of the target part is quite simple as it contains only two modules. The Concrete Execution (CE) core takes charge of debugging tested programs and getting run-time information. According to the commands received from the debug agent, the CE core can start and stop tested programs, set breakpoints, make single steps, and resume tested programs. The run-time information acquired includes the executed instructions, the concrete values of specified registers and memory locations, and so on. The debug agent is the unique interface in a target to communicate with a host. Just as its name implies, the debug agent redirects commands to the CE core, and sends run-time information to the host. Note that the two modules for communication, debugger and debug agent, are provided by the vendors of embedded systems instead of being developed by us.

B. Cooperation of Modules

The cooperation of all modules is described through a typical run of concolic execution. In the beginning, a tested program is uploaded from the host to the target through the debugger and the debug agent. Then a test input is chosen by the test selector out of the test pool. The test input is also uploaded to the target in the same way. In the first run, the test pool usually has only one test input which is generated randomly. After that, the CE core starts the tested program, while run-time information is returned to the host.

The SE core reads run-time information from the debugger. Note that symbolic execution is coordinated with concrete execution through cross-debugging functions. Once the tested program reads the test input, the SE core symbolizes the input. After symbolization, the two functions of tracking the propagation of symbols and generating the path condition, which are mainly reused from SMAFE (our previous concolic execution tool for general-purpose computers), are invoked.

When concrete execution stops (normally or abnormally such as a time out event), the gathered path condition is sent to the path selector. According to its path selection algorithm, one or more new path conditions will be constructed by the path selector. At last, the theorem prover generates new test inputs, and stores them in the test pool if the entered path conditions are satisfiable.

C. Advantages of the Architecture

The advantages of our method can be discerned from Fig. 2.

1) Our method overcomes software and hardware limitations of embedded systems, thus it becomes practical to test embedded software concolically in real devices. We claim that, taking advantage of the proposed architecture, any embedded system which supports remote debugging can be tested concolically regardless of its CPU performance and memory capacity.
2) Substantial manual work for code porting can be saved. Fig. 2 shows that symbolic execution including input symbolization, symbols tracking, generation of path conditions, constraint solving, path selection, etc., happens on hosts. So our method can reuse a great body of code directly from existing concolic execution tools for general-purpose computers. Therefore, our method is cheaper than existing methods [24]–[31] in porting effort. Even if we port a tool for x86-Windows to x86-VxWorks in implementation, our architecture is also applicable to IR-based concolic execution tools (explained in Section VI).
3) Accurate run-time information can be acquired. Fig. 2 reveals that concrete execution takes place in the targets (i.e., real devices). Therefore, the imperfection of simulators which could be a problem for simulation-based methods [30], [31] to get precise run-time information is undoubtedly not a problem for our method.
4) Our method does not require the source of tested software. Binary-level analysis is relevant in at least three situations [30], [31]: when no high-level source code is available, when the compiling process cannot be trusted, or when the increase of precision is essential. Hence, it is rewarding to conduct binary-level analysis.

V. Algorithm Description

Fig. 3 demonstrates the algorithm of the proposed method. The tested program P, and the initial test input t_0 are the inputs of the algorithm. The initial test input is always picked randomly. The output of the algorithm is a set of test inputs generated by our method.

The input set is initialized as empty in line 1, and then a work list *toExplore* is initialized as an empty set in line 3. The loop from line 4 repeatedly explores new paths of the tested program P until the work list becomes empty, which implies all paths are exercised, or one of the termination criterions is met. Typical termination criterions include a preset test time budget, a pre-established code coverage requirement, or uncaught exceptions, etc. The first procedure in the loop is to pick out a test input from the work list *toExplore*. Actually, the selection of test input depends on the path selection algorithm. For example, if a depth-first or breadth-first search algorithm is applied, the work list contains only one test input before the loop exits. However, for the generational algorithm [5], the test input with the highest coverage gain will be chosen.

Later, the tested program P with its test input t_0 will be executed concolically. Concolic execution is represented in the subroutine *concolic_execution* from line 14 to line 27. We will

```
parameteres: Program P, Initial test input t0
result: Test input set T
1   T = ∅;
2   toExplore = emptySet();
3   insert(toExplore, t0);
4   while not empty(toExplore) and not should_terminate() do
5       t = choose_one_input(toExplore);
6       pc = concolic_excution(P, t);
7       new_pcs = emptySet();
8       insert(new_pcs, negate_pc(pc));
9       new_inputs = emptySet();
10      insert(new_inputs, solve(new_pcs));
11      insert(toExplore, new_inputs);
12      insert(T, new_inputs);
13  return T;
14  Subroutine concolic_execution(P, t)
15  pc = ∅;
16  while not end_of(P) do
17      i = current_ins(P);
18      type = concrete_execution(i);
19      switch (type)
20          case DATA_INPUT:
21              input_symbolization(); break;
22          case DATA_TRANSFER:
23              symbol_propagation(); break;
24          case CONDITIONAL_BRANCH:
25              pc = pc ∧ generate_constraint(); break;
26          default: break;
27  return pc;
```

Fig. 3. Algorithm of the proposed method.

explain this subroutine later. After one run of concolic execution, a path condition pc is generated. Then a set of path conditions are generated by negating some of the constraints in the original path condition (lines 7, 8). The selection of negated constraints relies on a path selection algorithm. For example, a depth-first search algorithm will negate the constraint which is the farthest from the root, while a breadth-first algorithm will choose the nearest constraint. Differently, a generational algorithm will negate a group of constraints which will produce a set of new path conditions.

After that, all path conditions in the set *new_pcs* are solved by a theorem prover (STP in our prototype) in line 10. Some path conditions are not satisfiable, indicating the corresponding paths are not feasible, so no test inputs will be generated. Otherwise, new test inputs will be added to the work list *toExplore*, and also to the test input set T (lines 11, 12). Finally, the test set T will be outputted (line 13).

The subroutine *concolic_execution* is described in detail below. The inputs of the routine are the tested program P and its test input t. In the end, this routine outputs the path condition pc. In the very beginning, the path condition pc is set as empty in line 5. Then the program P is executed both concretely and symbolically (from line 16 to line 26). The first procedure of this loop is to pick out the current instruction i. Then, this instruction is executed concretely, and the type of this instruction is obtained (line 18).

Later on, different types of instructions will be executed symbolically according to their semantics. At a high level, all instructions are grouped into three categories. The instructions in the same category are handled in the same way as shown in Fig. 3. However, from the aspect of implementation, the instructions, even in the same category, are handled individually because their semantics are different. If the instruction indicates some values are read from the test input (line 20), then the related memory locations will be symbolized. If the instruction results in data flow (line 22), the symbolic store (where to store symbols) should be updated accordingly. So symbols will propagate from source operands to target operands after the symbolic execution of instruction i (line 23). If the instruction belongs to the conditional jump category, a new constraint will be produced, and the path condition will be updated as the conjunction of the original path condition and the new constraint (line 25). The path condition is satisfiable only if there exists a solution that satisfies both the left part and the right part of the mathematical conjunction notation $\bigwedge$.

The core algorithm of the proposed method is similar to the algorithm of SMAFE, except for the underlined lines in Fig. 3. For SMAFE, the underlined lines indicate that the associated procedures should interact with Pin. For example, the function *symbol_propagation* in line 23 requires concrete values of memory locations and registers by invoking the APIs provided by Pin if the associated operands are not symbols. However, for the proposed method, the underlined procedures should be adapted to embedded systems because we use cross-debugging functions to fetch concrete values. The porting effort is minimal for two reasons. First, the code without underlines can be reused directly, such as path selection, solving for path conditions. Second, a great amount of the code in the underlined procedures is also reusable because only a few lines of code will invoke Pin.

VI. Implementation

We implement our method on Wind River VxWorks (version 6.6). In this section, we focus on how to port our previous concolic execution tool SMAFE to VxWorks. SMAFE is designed for Windows executables on general-purpose computers, which is developed on top of Pin and STP. The total code amount of SMAFE is about 10 000 lines of C and C++ code without comments. In the work, the code we must port is minimal, just about 1500 lines of code without comments. Our prototype is able to execute machine code concolically because SMAFE analyzes machine code directly.

To port IR-based concolic execution tools, we need to add a module to translate machine code into IR statements. In many cases, the translation module is already available because we can use existing binary analysis software to do the work, such

```
BOOL INS_RepPrefix(INS ins)
{
    INS_STRUCT* pInsDasm = Search_ins_map(ins);
    if(pInsDasm == NULL)
        return FALSE;
    if(pInsDasm->ins_op[0] == 0xF3)
        return TRUE;
    else return FALSE;
}
```

Fig. 4. Implementation of INS_RepPrefix.

as Valgrind, Vine, and BAP. In the cases that the target architecture is not supported by any exiting software, our method shares the same shortcoming with IR-based concolic execution tools that we have to develop a module from scratch to perform translation.

In some of the worst cases, our method can still save porting effort for path selection, tracking and maintenance of symbols, solving of constraints, and more. It is straightforward to replace STP with other SMT solvers because constraint solving runs on the host. Besides, SMAFE has already formed constraints according to the standard input format of SMT solvers. So actually our prototype reuses the constraint solving module of SMAFE without any modifications.

Note that the implementation details presented in this section are specific to VxWorks, but the architecture and algorithm of our method are general, and applicable to other embedded systems. To reuse SMAFE, we just need to replace all invocations of Pin with equivalent functions by means of the WTX protocol. As an example, Fig. 4 shows the implementation of *INS_RepPrefix*. The function is actually provided in Pin. We have implemented the function by using the WTX protocol while keeping the invoking manner. The benefit is that our prototype can invoke this function as usual regardless of its inner implementations.

We detail the implementation of *INS_RepPrefix* as follows. The return value of *INS_RepPrefix* is true if the instruction *ins* has a REP ($0\times$F3) prefix. The instruction *ins* is fetched by invoking WTX APIs, and then the instruction is interpreted by an open-source disassemble library before the invocation of *INS_RepPrefix*. Then the function *Search_ins_map* is used to get the corresponding information (stored in *INS_STRUCT*) of this instruction. Besides, the machine code which is used to make judgment is stored in the array *ins_op* of *INS_STRUCT*. The example demonstrates the easiness of porting by our method.

Table I presents the major WTX APIs invoked by our prototype for communication as well as their descriptions. The rest of this section presents specific issues of VxWorks rather than the topics on general-purpose computers which can be found in our recent paper [33].

TABLE I
WTX APIs Invoked by Our Prototype

	API	Description
1	wtxObjModuleLoad	Upload tested programs and test inputs to the target.
2	wtxObjModuleInfoGet	Get the location where the tested program loaded. The location information is used to discriminate the instructions belonging to the tested program or the environment.
3	wtxRegsGet	Get concrete values of specified registers.
4	wtxMemRead	Get concrete values of specified memory locations.
5	wtxMemDisassemble	Disassemble the specified instructions. But the returned information is inconvenient for symbolic execution. We then interpret the instructions using an open-source disassembler.
6	wtxContextResume	Resume execution of the tested program. There are two places to invoke the API. The first is to start the tested program because the loaded program is suspended in default. The second is to run the external code concretely.
7	wtxEventpointAdd	Set breakpoints by setting the event as TEXT_ACCESS.
8	wtxContextCont	Continue execution stopped at a breakpoint.
9	wtxRegisterForEvent	Register callbacks for events. Our prototype handles two events (*i.e.*, TEXT_ACCESS, CTX_EXIT) specially (see Subsection VI-B).

A. Initialization

The initialization of SMAFE is quite straightforward, taking advantages of Pin. However, we have to initialize concolic execution step by step in the proposed prototype because some necessary functions are not directly provided in the WTX protocol. For example, given the Pin, we can register a callback function which will be called before the execution of each instruction via registering an instrumentation function by invoking *INS_AddInstrumentFuntion*, and then registering the callback function by calling *INS_InsertCall* in the aforementioned instrumentation function. However, to achieve the same function by the WTX protocol, we first set a breakpoint on each instruction, and then handle TEXT_ACCESS events. The initialization procedure is detailed as follows.

1) Create a WTX session.
2) Connect to the target.
3) Register a function to handle all events. Actually, only two kinds of events are specially dealt with. The first is TEXT_ACCESS, indicating a breakpoint is reached. The other is CTX_EXIT, representing the tested program exits.
4) Upload the tested program and the initial input to the target. The location of the tested program in the target main memory is recorded to discriminate the instructions in the tested program from those in other modules of VxWorks (see Subsection VI-D).
5) Look for the entry point of the tested program, and set a breakpoint on the entry point. Hence, when the entry point is reached, a TEXT_ACCESS event will be triggered.
6) Callback functions are registered, which will be called before the execution, and after the return of Input-Output (I-O) functions. Those functions cooperate to symbolize test inputs (see Subsection VI-D).
7) Run the tested program, and wait for the notifications of events.

B. Events Handling

When a TEXT_ACCESS event is triggered, our prototype handles the event as follows.

1) Delete the breakpoint on this instruction.
2) Get the concrete values of all registers, and store them in a data structure.
3) Get the current instruction, and interpret it using a disassemble library. The detailed information is stored in a data

structure for future use, such as the tracking of symbols, generating path conditions, etc.

4) Judge whether this instruction is the first instruction of a function (denoted by $f1$).
   If so, our prototype looks for whether there is a registered callback function (denoted by $f2$) that should be invoked before the execution of this instruction. If so, our prototype invokes $f2$. And then our prototype sets a breakpoint on the first instruction which will be executed after the execution of $f1$.
5) Judge whether this instruction is the first instruction after the execution of a function. If so, our prototype looks for whether there is a registered callback function (denoted by $f1$) that should be called before its execution. If so, our prototype calls $f1$.
6) Judge whether this instruction should be executed symbolically (see Subsection of VI-D). If so, the code for symbolic execution (mostly reused from SMAFE) will be executed.
7) Single-step the current instruction if it has been executed symbolically; otherwise, resume the tested program. Note that the instruction which will be executed after the current instruction will be broke (i.e., a TEXT_ACCESS event will be triggered automatically) by stepping through tested programs instruction by instruction.

The event-handling function for event CTX_EXIT takes charge of the work when the tested program terminates, such as releasing any internal storage used by the WTX protocol, unloading the tested program, and releasing the storage used by symbolic execution. The coding of this function should be careful to avoid memory leakage.

C. Symbolization of Inputs

To monitor and symbolize test inputs, the prototype registers several functions which will be invoked before and after the execution of I-O functions. The intuition is that inputs are usually read via I-O functions provided by I-O libraries or operating systems. Here we only present the handling of two I-O functions: *fopen* and *fread*. The procedure of symbolization of inputs is as follows.

1) Exactly before the execution of *fopen*, the registered callback function *fopen_before* is invoked. *fopen_before* is used to record the name of the opened file.
2) Exactly after the execution of *fopen*, the registered callback function *fopen_after* is invoked. *fopen_after* first checks whether the file is opened successfully. If so, the pointer to the file returned by *fopen* is associated to the file name which has been recorded in step 1.
3) The registered callback function *fread_before* is executed exactly before the execution of *fread*. *fread_before* first looks for the name of the operated file according to its pointer, and then checks whether the file is the test input. If so, the address of the buffer which is used to store the content of the test input and the item size in bytes are recorded.
4) Exactly after the execution of *fread*, the registered callback function *fread_after* is called. *fread_after* first examines whether the test input is read successfully by checking the return value of *fread*. If so, the return value indicating the number of items actually read, the item size, and the address of the buffer are associated.

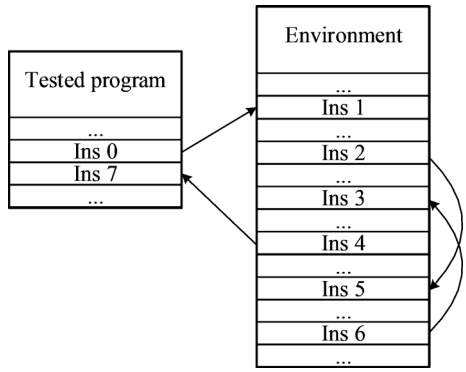

Fig. 5. An example to describe the procedure of environment handling.

5) The remaining steps, such as creating new items in symbolic store, and allocating symbols ('input1', 'input2', . . ., 'inputN') to the inputs according to their locations in the test input, have been implemented in the reused code of SMAFE.

The invocation of the callback functions depends on the event-handling procedure for TEXT_ACCESS events (Step 4 and Step 5 of the steps in Subsection VI-B).

D. Environment Handling

Typical embedded software in practice will interact with the other modules of embedded systems (i.e., environment), so a good concolic execution tool should handle the environment properly. There are two popular modes to do this work: Strictly Consistent Unit-level Execution (SC-UE), and Strictly Consistent System-level Execution (SC-SE) [35]. According to their definitions, the SC-UE mode executes the code in the environment concretely, while the SC-SE mode executes external code concolically.

In our prototype, we use the SC-UE mode at the cost of accuracy because the SC-UE mode is significantly faster than the SC-SE mode in handling external code. The procedure of environment handling is described as follows, referring to Fig. 5.

In Fig. 5, the left box indicates the tested program, the right one stands for external code, each *Ins* means an instruction in the tested program or in the environment, and each arrow indicates a control flow transfer. Usually, control flow transfers between the tested program and the environment are caused by the calls or the returns of external functions. So the instruction located exactly after the call to the environment will be executed after the execution of external code.

When *Ins* 1 is encountered, the callback function to handle TEXT_ACCESS events will be invoked. The callback function first examines whether *Ins* 1 should be executed symbolically. According to the SC-UE mode, the answer to the above question is no because *Ins* 1 does not belong to the tested program. Then the instruction (actually it is *Ins* 0 in Fig. 5) which has been executed just before *Ins* 1 is identified. Afterwards, the callback function looks for the instruction located just after *Ins* 0, i.e., *Ins* 7 in Fig. 5, and sets a breakpoint on it. Finally, the tested program is resumed.

The mentioned procedure is simple but efficient. Another benefit of the procedure is that it is able to handle complex situations in practice. For example, in most cases, external

TABLE II
EXPERIMENTAL ENVIRONMENT

	Host	Target
CPU	Pentium Dual-Core E5400	Pentium IV 1.6GHz
Memory	2GB	256MB
OS	Windows 7	VxWorks 6.6

functions are nested, but it is not a problem to correctly distinguish external code based on the proposed handling procedure. Distinguishing is not a problem because the prototype sets a breakpoint on the instruction in the tested program that will be executed exactly after the execution of the environment, before control flow transfers.

VII. CASE STUDY

The purpose of this section is to verify the feasibility of our method in practice through three case studies. The first case study consists of two internal modules of VxWorks. The first is *qsort* which implements a quick-sort algorithm to sort an array, and the other is *bsearch* which performs a binary search of a sorted array. Before the presentation of results, we firstly introduce the experimental environment in Table II. The host is a personal computer, and the target is an embedded system with self-owned intellectual property rights. The target is connected with the host through serial ports.

The basic block coverage of the tested program achieves around 80% (99 out of 123) after 27 paths are explored by our prototype. The instruction coverage, and branch coverage are 81% (468 out of 578), and 77.3% (99 out of 128) respectively. We use the metric of instruction coverage, rather than statement coverage, because we test binaries without debugging information. We have to note that we do not count the basic blocks, instructions, and branches belonging to external code. The time consumption for this experiment is quite short, 22.7 seconds in total.

Then we analyze the reasons for the uncovered basic blocks through manual reverse engineering. We do not analyze the reasons for uncovered instructions because instruction coverage is equivalent to basic block coverage in essence. We also do not present the reasons for uncovered branches just because the analysis process is quite similar with that for uncovered basic blocks. We find that all the uncovered basic blocks belong to *qsort*. In other words, *bsearch* is tested thoroughly.

Reverse engineering reveals the internal logic of *qsort*: if the number of elements in the entered array is smaller than 2, sort is not needed, and then *qsort* exits; if the number is 2 or 3, the function *insertion_sort* is called; otherwise, the function *quick_sort* is invoked. Therefore, in the default situation, all basic blocks in *insertion_sort* are not executed as the length of inputs is much larger than 3. We then change the experimental setting by fixing the length of the inputs to 3, and then rerun the experiment. Results validate the analysis above that all uncovered basic blocks are executed now.

The second case study involves *inflateLib* which is used to inflate a compressed data stream, primarily for boot ROM decompression. In typical execution, *inflateLib* will be invoked to decompress the executable in compressed boot ROMs, and then jump to it. The goal of this case study is to demonstrate the amount of code that could be covered by our prototype. The size of the tested program is nontrivial, actually 582 basic blocks (3271 instructions) in total. The basic block coverage achieves about 57.2% in 34 minutes, and no more basic blocks can be executed from then on. The instruction coverage, and branch coverage are 60% (1963 out of 3271), and 50% (302 out of 604), respectively.

We do not explore the reasons for uncovered basic blocks thoroughly because manual reverse engineering for such a complex program is very expensive. A quick but incomplete manual analysis shows that concolic execution is hindered by complex mathematical algorithms (e.g., lz77) applied by *inflateLib*, as well as complicated internal data structures (e.g., hash table, Huffman tree) used by the tested program.

The last case study involves the shell program of VxWorks with the main function of parsing shell commands. It is a relatively large program which contains 855 basic blocks (5651 instructions). The basic block coverage achieved by our prototype is about 61.3% when the preset testing time (i.e., 12 hours in this case) is exhausted. The instruction coverage, and branch coverage are 62.8% (3549 out of 5651), and 54.1% (477 out of 882) respectively. Due to the complicated internal logic of the tested program, we also do not explore the reasons for uncovered basic blocks completely. But we still observe several hints that our prototype might run into path explosion when testing the shell program.

The first observation is that the number of paths of the shell program is so large that our prototype cannot explore each of them in 12 hours. Second, quite a number of generated inputs are syntactically invalid, so in those runs, the tested program quits with only the code in the very beginning stages of the parsing process being executed. Therefore, we could have expected higher code coverage if more testing time was available. But we believe a wiser suggestion would be to provide an input grammar to ensure all generated inputs by concolic execution are syntactically valid.

We have to note that the last two tested programs are very large for concolic testing. Both of them have more than 500 basic blocks, so the source code should contain thousands of lines (VxWorks is not open-source). Recent work stated that the upper limit of concolic testing is a few thousands lines [49]. Even in a recent empirical study, the tested programs which are selected from several practical software packages range from 6 to 214 lines of code [50]. To promote the scalability of concolic execution, we believe traditional approaches (e.g., intelligent path selection algorithm [5], [23], [48], summaries [6], symbolic grammar [51], [52]) are insufficient. We have to resort to parallel computing [53], [54] to utilize massive computing and memory resources.

The three case studies above validate our claim that our method is feasible to apply to embedded software in practice. But we also concern another question which cannot be answered by the above case studies: does our method for porting bring about negative impacts on both effectiveness and efficiency for concolic execution? This question will be answered in the next section.

VIII. EVALUATIONS ON BENCHMARKS

A. Experiments Setup

In this section, we investigate the reasons for uncovered basic blocks by testing ten programs from a benchmark SGLIB [55].

TABLE III
EXPERIMENTAL RESULTS STATISTICS

Tested program	Total BBLs	Executed BBLs	BBLs%	Total Ins	Executed Ins	Ins%	Total Brs	Executed Brs	Brs%
dllist	100	96	96	325	316	97.2	91	85	93.4
listsort	40	40	100	133	133	100	35	31	88.6
list insert sort	18	18	100	163	163	100	20	18	90
list insert sort1	34	28	82.4	113	92	81.4	29	24	80.1
array bin search	25	25	100	154	154	100	26	24	92.3
array sort	52	52	100	171	171	100	48	47	97.9
array sort1	43	43	100	187	187	100	50	49	98
hash	58	54	93.1	345	313	90.7	61	56	91.8
rbtree	128	101	78.9	673	519	77.1	154	121	78.6
queue	48	43	89.6	278	244	87.8	52	40	76.9
Average	**54.6**	**50**	**91.6**	**254.2**	**229.2**	**90.2**	**56.6**	**49.5**	**87.5**

Then the efficiency of the prototype which is measured by time consumption is presented. We have to analyze the causes of uncovered basic blocks by inspecting the source, and reverse engineering, so the scales of the tested programs are relatively small (testing for larger programs have been shown in Section VII). Note that all tested programs are compiled by the compiler provided by Workbench IDE. So those programs run on the target rather than the host. Besides, the experimental environment is also built up according to Table II.

B. Code Coverage Analysis

Experimental results are shown in Table III. The basic block coverage, instruction coverage, and branch coverage are presented in column 4, 7, and 10 respectively. Finally, the last row calculates the average values of all tested programs.

Note that the numbers of total basic blocks, total instructions, and total branches are counted by IDA Pro [56]. In many cases, the total number of branches is more than the total number of basic blocks, but there are exceptions. Because each control flow transfer (e.g., conditional jumps, unconditional jumps, function calls, function returns) may increase the count of basic blocks while only conditional jumps are treated as branches. As in the last section, we do not take the basic blocks, instructions, and branches of external code into account. Besides, repeated executed basic blocks, instructions, and branches are counted only once.

Table III reports that our prototype achieves high code coverage such that the average basic block coverage of the ten tested programs is 91.6%, which is much higher than the results from testing large-scale programs (see Section VII). The result that the instruction coverage of each program is in accordance to basic block coverage is as expected. The reason is that basic block coverage is essentially the same metric as instruction coverage, except that a basic block consists of one or more sequential instructions. Another observation from Table III (the last row) is that the average branch coverage of the ten tested programs is no larger than the basic block coverage and the instruction coverage because the branch coverage is stricter than the other two in principle. Even so, our prototype performs quite well when it is evaluated by branch coverage (87.5% on average).

The latter part of this subsection analyzes the reasons for the uncovered basic blocks by our prototype. In the end of this subsection, we will answer the question about whether or not our method results in a negative influence on code coverage. For similar reasons, we do not present the reasons for uncovered instructions and uncovered branches here.

```
1:    if (it->subcomparator != NULL) {
2:        eq = it->equalto;
3:        scp = it->subcomparator;
4:        while (ce!=NULL && scp(eq, ce)!=0)
5:            ce = ce->previous;
    }
```

Fig. 6. The code in statements 2 to 5 cannot be executed in dllist.

```
1:    if (it->subcomparator != NULL) {
2:        eq = it->equalto;
3:        scp = it->subcomparator;
4:        while (ce!=NULL && (c=scp(ce, eq)) < 0)
5:            ce = ce->next;
6:        if (ce != NULL && c > 0)
7:            ce = NULL;
    }
```

Fig. 7. The code in Statements 2 through 7 cannot be executed in list insert sort 1.

dllist—4 basic blocks out of 100 are not covered. Analysis reveals that the statements, in Fig. 6, from 2 through 5 which consist of 4 basic blocks, cannot be executed. The reason lies in that the predicate in Statement 1 is always evaluated as false regardless of test inputs. In fact, the 'it->subcomparator' is preseted as NULL in the function *sglib_dllist_it_init*. As a result, those 4 basic blocks cannot be covered through changing inputs.

list insert sort1—6 basic blocks out of 34 are not covered. Statement 2 through Statement 7 in Fig. 7, which consist of 6 basic blocks, are not executed because the predicate in Statement 1 is evaluated to false under all conditions. Actually, the 'it->subcomparator' is fixed to NULL in the function *sglib_iListType_it_init*. So those 6 basic blocks cannot be executed in any case.

hash–4 basic blocks out of 58 are not executed due to the same reason as that of dillist.

rbtree–27 basic blocks out of 128 are not executed. Specifically, one basic block is not covered because the predicate in the macro 'assert(it! = NULL)' is evaluated as true under all conditions. Analysis shows that the 'it' part of the macro is a pointer which is assigned as a non-zero parameter of the function *sglib_rbtree_it_init*. So that basic block cannot be covered in any case.

Fig. 8 presents 13 uncovered basic blocks (Statement 4 through Statement 7). The reason lies in the predicate in Statement 1 because it is evaluated as true under all conditions. Analysis also indicates that the 'equalto' is fixed to NULL in the function *sglib_rbtree_it_init*. The other 13 basic blocks are not executed due to a similar reason as that in Fig. 8, the

```
1:      if (equalto == NULL) {
2:          t = tree;
3:      }else {
4:          if (subcomparator == NULL) {
5:              SGLIB___BIN_TREE_FIND_MEMBER
(type, tree, equalto, left, right, comparator, t);
6:          }else {
7:              SGLIB___BIN_TREE_FIND_MEMBER
(type, tree, equalto, left, right, subcomparator, t);
            }
        }
```

Fig. 8. The code in Statements 4 to 7 cannot be executed in rbtree.

variable 'equalto' is replaced with 'eqt', and its value is also fixed to NULL. Therefore, we find those 27 basic blocks are not able to be covered due to the program logic.

queue—5 basic blocks out of 48 are not executed. Among them, one basic block in the function *SGLIB_QUEUE_ADD_NEXT* is not executed because the length of inputs is shorter than the capacity of the queue. In other words, the basic block can be covered by setting the length of inputs properly. To this end, we have to change the length manually in our experimental setting because our prototype does not treat the length of inputs as a symbol. As a result, the length of inputs is fixed during concolic execution.

One basic block in the function *SGLIB_HEAP_ADD*, and one in the function *SGLIB_HEAP_ADD_NEXT* (invoked by *SGLIB_HEAP_ADD*) are not executed because the length of inputs is shorter than the capacity of the heap. However, if we set the length of inputs equal to the capacity of the heap, then the tested program will exit before the execution of the function *SGLIB_HEAP_ADD* due to the program logic. As a consequence, those two basic blocks cannot be covered regardless of the test inputs.

One basic block in the function *SGLIB_QUEUE_DELETE_FIRST*, and one in the function *SGLIB_HEAP_DELETE_FIRST* are not covered because the queue and the heap are not empty. However, the queue and the heap are ensured to be not empty before the invoking of the two functions due to the program logic. Therefore, the two basic blocks cannot be executed in any case.

As a conclusion, only one of those uncovered basic blocks is able to be executed by adjusting the length of inputs properly, and the others cannot be executed in any case due to the program logic. In the end, the answer for the question raised in the 4th paragraph of SubSection VIII-B is that the porting work of our prototype has no negative impacts on code coverage.

C. Performance Analysis

In this experiment, we analyze the average time spent in one run of concolic execution, and the results are shown in Table IV. In this subsection, we will answer the question of whether our method for porting leads to negative influence on the efficiency of concolic execution. The last column computes the time used per instruction. The number of executed instructions not only relies on tested programs but also test inputs. For example, if we feed the simple bubble sort program with long inputs, the number of executed instructions should be very large. Conclusion can be drawn from Table IV that the efficiency of our prototype is acceptable.

TABLE IV
TIME CONSUMPTION FOR ONE RUN OF CONCOLIC EXECUTION (IN MS)

Tested program	Ins	Time	Time/Ins
dlllist	1711	3391	1.98
listsort	1000	2000	2.00
list insert sort	486	1297	2.67
list insert sort1	765	1875	2.45
array bin search	710	1422	2.00
array sort	599	1359	2.27
array sort1	1031	2051	1.99
hash	3436	5656	1.65
rbtree	3619	5610	1.55
queue	823	1547	1.88

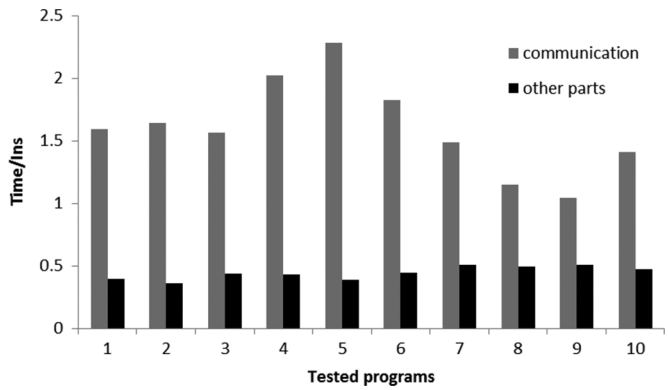

Fig. 9. Time consumptions per instruction for communication, and the other parts of concolic execution.

Fig. 9 visualizes the time consumptions for communication between a host and a target, as well as the other parts (constraint solving etc.). The numbers from 1 to 10 in the x-axis of Fig. 9 correspond to tested programs from row 2 to row 11 in Table IV. An obvious observation from Fig. 9 is that the time spent for communication accounts for a large proportion of total time consumption. More precisely, the average time for communication is around 3 times longer than the time for the other parts. So the answer to the question raised in the first paragraph of this subsection is that the porting work of our prototype has visible negative impact on the efficiency of concolic execution due to communication cost. But the long communication time can be reduced. We claim that the long communication time does not result from our architecture and algorithm; instead, it lies in our implementation (see Section IX).

IX. DISCUSSION

This work proposes a novel method to test embedded software concolically. It is an alternative to existing methods, which are based either on simulation or on instrumentation. Even though our method possesses several attractive advantages, we do not claim it can solve every issue related to this research topic. This section discusses the limitations of our method, as well as possible solutions for future research.

The first kind of limitation results from the design of our method. Specifically, our method is intrusive, which will dramatically slow down the execution of tested software that runs on the embedded systems.

So our method cannot be expected to perform well in testing time-critical software. To address the problem, we plan to design a less intrusive method in our future work.

The second kind of limitation is incurred by the implementation of our prototype. To be specific, our implementation incurs relatively long communication time. Our implementation coordinates symbolic execution and concrete execution synchronously. But obviously, symbolic execution runs much slower (at least one order of magnitude) than concrete execution. So the concrete execution part on the target has to wait for the completion of the symbolic execution part on the host. We plan to coordinate symbolic execution and concrete execution asynchronously in future work. For asynchronous mode, all run-time information is returned to a host quickly, and then stored in a cache on the host. SE core will fetch run-time information from the cache rather than from the target directly. So the communication time for the asynchronous mode is expected to be minimal.

The last kind of limitation contains the common issues encountered by general-purpose computers, as well as embedded systems, including path explosion, pointers, native calls, string operations, non-linear arithmetic, function pointers, and floating-point operations. Details about those challenges and associated solutions can be found in our survey [57], and a recent empirical study [50].

X. Conclusion

In this work, we propose a concolic-execution-based method to automatically generate test inputs for embedded software in a real environment. The core idea of our method is dividing concolic execution into symbolic execution on the hosts, and concrete execution on the targets. Our method overcomes the harsh hardware and software constraints of embedded systems because all heavy-weight work is constrained on resourceful hosts. The host communicates and coordinates with the target via cross-debugging functions provided by the vendors of embedded systems. Significant porting effort can be saved due to our architecture.

We then implement a prototype on VxWorks. Case studies in practice validate the feasibility of our method. Experiments demonstrate that our prototype achieves high code coverage with acceptable speed. As far as we know, we are the first to apply concolic execution to VxWorks.

References

[1] J. C. King, "Symbolic execution and program testing," *J. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[2] G. J. Myers, *The Art of Software Testing*. New York, NY, USA: Wiley, 1979.

[3] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proc. Joint 10th Eur. Software Engineering Conf. (ESEC-10) and 13th ACM SIGSOFT Symp. Foundations of Software Engineering (FSE-13)*, 2005, pp. 263–272.

[4] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2005, pp. 213–223.

[5] P. Godefroid, M. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proc. Network and Distributed System Security Symp.*, 2008, vol. 8, pp. 151–166.

[6] P. Godefroid, "Compositional dynamic test generation," in *Proc. 34th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, 2007, pp. 47–54.

[7] Fuzzgrind: An Automatic Fuzzing Tool, May 2013 [Online]. Available: http://esec-lab.sogeti.com/dotclear/index.php?pages/Fuzzgrind

[8] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *Proc. IEEE Symp. Security and Privacy*, 2006, pp. 2–16.

[9] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Theory and techniques for automatic generation of vulnerability-based signatures," *IEEE Trans. Depend. Secure Comput.*, vol. 5, no. 4, pp. 224–241, 2008.

[10] D. Brumley, H. Wang, S. Jha, and D. Song, "Creating vulnerability signatures using weakest preconditions," in *Proc. IEEE Computer Security Foundations Symp.*, 2007, pp. 311–325.

[11] Z. K. Liang and R. Sekar, "Fast and automated generation of attack signatures: A basis for building self-protecting servers," in *Proc. ACM Conf. Computer and Communications Security*, 2005, pp. 213–222.

[12] J. Newsome, D. Brumley, D. Song, J. Chamcham, and X. Kovah, "Vulnerability-specific execution filtering for exploit prevention on commodity software," in *Proc. Network and Distributed System Security Symp.*, 2006.

[13] D. Brumley, C. Hartwig, M. G. Kang, Z. K. Liang, J. Newsome, P. Poosankam, and D. Song, Bitscope: Automatically Dissecting Malicious Binaries. Tech. Rep. CS-07-133, 2007.

[14] D. Brumley, C. Hartwig, Z. K. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection, vol. 36 of Countering the Largest Security Threat Series: Advances in Information Security*. New York, NY, USA: Springer-Verlag, 2008.

[15] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Proc. IEEE Symp. Security and Privacy*, 2007, pp. 229–243.

[16] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, J. R. Vijay, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2005, pp. 190–200.

[17] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Proc. Conf. Compiler Construction*, 2002, pp. 213–228.

[18] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[19] D. M. Leonardo and B. Nikolaj, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Germany: Springer, 2008, pp. 337–340.

[20] Apr. 2013, STP (Simple Theorem Prover) [Online]. Available: http://sourceforge.net/projects/stp-fast-prover

[21] May 2014, lpsolve [Online]. Available: http://sourceforge.net/projects/lpsolve/

[22] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proc. 23rd IEEE/ACM Int. Conf. Automated Software Engineering*, 2008, pp. 443–446.

[23] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. USENIX Symp. Operating System Design and Implementation*, 2008, vol. 8, pp. 209–224.

[24] Y. Kim and M. Kim, "SCORE: A scalable concolic testing tool for reliable embedded software," in *Proc. 19th ACM SIGSOFT Symp. and the 13th Eur. Conf. Foundations of Software Engineering*, 2011, pp. 420–423.

[25] Y. Kim, M. Kim, and Y. Jang, "Concolic testing on embedded software—case studies on mobile platform programs," in *Proc. European Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE) Industrial Track*, 2011.

[26] Y. Kim, M. Kim, and Y. Jang, "Industrial application of concolic testing on embedded software: Case studies," in *Proc. IEEE Int. Conf. Software Testing, Verification and Validation*, 2012, pp. 390–399.

[27] M. Kim, Y. Kim, and Y. Kim, "Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE," in *Proc. Int. Conf. Software Engineering*, 2012, pp. 1143–1152.

[28] Y. Kim, M. Kim, and N. Dang, "Scalable distributed concolic testing: A case study on a flash storage platform," in *Proc. Int. Conf. Theoretical Aspects of Computing*, 2010, pp. 199–213.

[29] Y. Kim and M. Kim, "Concolic testing of the multi-sector read operation for flash memory file system," in *Proc. Brazilian Symp. Formal Methods: Foundations and Applications*, 2009, pp. 251–265.

[30] S. Bardin and P. Herrmann, "Structural testing of executables," in *Proc. IEEE Int. Conf. Software Testing, Verification and Validation*, 2008, pp. 22–31.

[31] S. Bardin and P. Herrmann, "OSMOSE: Automatic structural testing of executables," *Softw. Test., Verif. Rel.*, vol. 21, no. 1, pp. 29–54, 2009.

[32] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The BINCOA framework for binary code analysis," *Comput. Aided Verif.*, pp. 165–170, 2011.
[33] T. Chen, X. Zhang, C. Zhu, X. Ji, S. Guo, and Y. Wu, "Design and implementation of a dynamic symbolic execution tool for Windows executables," *J. Softw.: Evol. Process*, vol. 25, no. 12, pp. 1249–1272, 2013.
[34] D. A. Molnar and D. Wagner, Catchconv: Symbolic Execution and Run-Time Type Inference for Integer Conversion Errors University of California, Berkeley, CA, USA, Tech. rep., 2007, pp. 2007–2023.
[35] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *Comput. Architect. News*, vol. 39, no. 1, pp. 265–278, 2011.
[36] D. Song, D. Brumley, Y. Heng, J. Caballero, I. Jager, G. K. Min, Z. K. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *Information Systems Security*. Berlin, Germany: Springer, 2008, pp. 1–25.
[37] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proc. IEEE Symp. Security and Privacy*, 2012, pp. 380–394.
[38] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Computer Aided Verification*. Berlin, Germany: Springer, 2011, pp. 463–469.
[39] K. G. Larsen, M. Mikucionis, and B. Nielsen, "Online testing of realtime systems using UPPAAL," in *Proc. 4th Int. Workshop Formal Approaches to Software Testing*, 2004, pp. 79–94.
[40] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing real-time embedded software using UPPAAL-TRON: An industrial case study," in *Proc. 5th ACM Int. Conf. Embedded Software*, 2005, pp. 299–306.
[41] C. Nebut, F. Fleurey, Y. Le Traon, and J. M. Jezequel, "Automatic test generation: A use case driven approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 140–155, 2006.
[42] L. Lazic and D. Velasevic, "Applying simulation and design of experiments to the embedded software testing process," *Softw. Test., Verif. Rel.*, vol. 4, no. 4, pp. 257–82, 2004.
[43] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival, "Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software," *Essence Computat., Lecture Notes Comput. Sci.*, vol. 2566, pp. 85–108, 2002.
[44] T. Chen, X. Zhang, X. Xiao, Y. Wu, and C. Xu, "SEVE: Symbolic execution based vulnerability exploring system," *Int. J. Computat. Math. Electron. Eng.*, vol. 32, no. 2, pp. 620–637, 2013.
[45] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: A system for automatically generating inputs of death using symbolic execution," in *Proc. ACM Conf. Computer and Communications Security*, 2006.
[46] R. G. Xu, P. Godefroid, and R. Majumdar, "Testing for buffer overflows with length abstraction," in *Proc. Int. Symp. Software Testing and Analysis*, 2008, pp. 27–37.
[47] T. L. Wang, T. Wei, G. F. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proc. IEEE Symp. Security and Privacy*, 2010, pp. 497–512.
[48] N. Tillmann and J. De Halleux, "Pex-white box test generation for .NET," in *Proc. 2nd Int. Conf. Tests and Proofs*, 2008, pp. 134–153.
[49] L. Ciortea, C. Zamfir, S. Bucur, S. Chipounov, and G. Candea, "Cloud9: A software testing service," *Operat. Syst. Rev.*, vol. 43, no. 4, pp. 5–10, 2009.
[50] X. Qu and B. Robinson, "A case study of concolic testing tools and their limitations," in *Proc. Int. Symp. Empirical Software Engineering and Measurement*, 2011, pp. 117–126.
[51] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2008, pp. 206–215.
[52] R. Majumdar and R. G. Xu, "Directed test generation using symbolic grammars," in *Proc. ACM/IEEE Int. Conf. Automated Software Engineering*, 2007, pp. 134–143.
[53] M. Staats and C. S. Pasareanu, "Parallel symbolic execution for structural test generation," in *Proc. Int. Symp. Software Testing and Analysis*, 2010, pp. 183–193.
[54] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proc. Eurosys Conf.*, 2011, pp. 183–197.
[55] Dec. 2013, SGLIB—A Simple Generic Library For C [Online]. Available: http://sglib.sourceforge.net/
[56] May 2014, The IDA Pro Disassembler and Debugger [Online]. Available: http://www.hex-rays.com/idapro/
[57] T. Chen, X. Zhang, S. Guo, H. Li, and Y. Wu, "State of the art: Dynamic symbolic execution for automated test generation," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1758–1773, 2013.

Ting Chen received the B.S. degree in mathematics, the M.S. degree, and the Ph.D. degree in computer science from the University of Electronic and Technology of China (UESTC), Chengdu, in 2007, 2010, and 2013, respectively.

Now he is a lecturer in compeer science at UESTC. He has published several articles in prestigious journals and conferences in the fields of malware analysis, intrusion detection, and software testing. He has obtained several Chinese patents with regard to computer security. His current research includes software reliability, software test case generation, and software verification.

Xiao-Song Zhang received the B.S. degree in dynamics engineering from Shanghai Jiaotong University, Shanghai, in 1990 and the M.S. and Ph.D. degrees in computer science from the University of Electronic and Technology of China (UESTC), Chengdu, in 2011.

Now he is a Professor in computer science at UESTC. He has worked on numerous projects in both research and development roles. These projects include device security, intrusion detection, malware analysis, software testing, and software verification. He has coauthored a number of research papers on computer security. His current research involves software reliability, software vulnerability discovering, software test case generation, and reverse engineering.

Xiao-Li Ji received the B.S. degree in network engineering from Southwest University (SWU), Chongqing, in 2010 and the M.S. degree in computer science from the University of Electronic and Technology of China (UESTC), Chengdu, in 2013.

She has obtained several Chinese patents with regard to software test case generation. Her current research includes software reliability, software test case generation, and software verification.

Cong Zhu received the B.S. and M.S. degrees in computer science from the University of Electronic and Technology of China (UESTC), Chengdu, in 2010 and 2013, respectively.

His current research includes software reliability, software test case generation, and software verification.

Yang Bai received the M.S. degree in computer science from the University of Electronic and Technology of China (UESTC), Chengdu, in 2013.

She now is an engineer in No. 30 Research Institute of China Electronics Technology Group Corporation (CETC). Her research focuses on information security and software security.

Yue Wu is a Professor at the University of Electronic and Technology of China (UESTC), Chengdu. He had served as Dean of the School of Computer Science and Engineering, the School of Information and Software Engineering, and the School of Software in Chengdu College of UESTC. He has published more than 70 research papers and authored 3 books. His current research focuses on grid computing, database system M.S., and data mining.

Prof. Wu is a committee member of several journals including *Journal of UESTC*, *Journal of Computer Applications*, and *Software World*. He has presided over a number of international conferences such as CIDE 2005 and ISNN 2006.