Security Testing of Software on Embedded Devices Using x86 Platform[*]

Yesheng Zhi(✉), Yuanyuan Zhang, Juanru Li, and Dawu Gu

Lab of Cryptology and Computer Security,
Shanghai Jiao Tong University, Shanghai, China

Abstract. Security testing of software on embedded devices is often impeded for lacking advanced program analysis tools. The main obstacle is that state-of-the-art tools do not support the instruction set of common architectures of embedded device (e.g., MIPS). It requires either developing new program analysis tool aiming to architecture or introducing many manual efforts to help security testing. However, re-implementing a program analysis tool needs considerable amount of time and is generally a repetitive task. To address this issue efficiently, our observation is that most programs on embedded devices are compiled from source code of high level languages, and it is feasible to compile the same source code to different platforms. Therefore, it is also expected to directly translate the compiled executable to support another platform. This paper presents a binary translation based security testing approach for software on embedded devices. Our approach first translates a MIPS executable to an x86 executable leveraging the LLVM-IR, then reuses existing x86 program analysis tools to help employ in-depth security testing. This approach is not only efficient for it reuses existing tools and utilizes the x86 platform with higher performance to conduct security analysis and testing, but also more flexible for it can test code fragment with different levels of granularity (e.g., a function or an entire program). Our evaluation on frequently used data transformation algorithms and utilities illustrates the accuracy and efficiency of the proposed approach.

Key words: Security testing, Binary translation, Embedded device, Binary analysis

1 Introduction

Security and privacy is considered as a significant requirement in embedded systems, especially considering that most of them are provided to system networks, private networks, or the Internet. However, the specialization of embedded system often comes with one or more inherent characteristics [9, 10], which make security analysis and testing on embedded systems significantly stricter than on traditional commodity systems.

[*] partially supported by Major program of Shanghai Science and Technology Commission (Grant No: 15511103002)

One way to employ security analysis is to leverage hardware debugging interface. This way requires dedicated hardwares and extreme resource requirements, therefore it can not be widely used in off-the-shelf embedded devices. To address, another way is leveraging dynamic binary translation (e.g., QEMU [4], Avatar [13]) to simulate and execute the binary program on PC system. The main drawback of dynamic binary translation is the large overhead of runtime translation and runtime optimization. Furthermore, for short-running programs, especially for interactive applications that are common on mobile devices, start-up time and response time are critical to their performances.

Consider that most state-of-the-art researches only focus on the x86 instruction set, and implement mature security tools. However, a embedded system usually uses specific instruction set architectures (ISAs), e.g., MIPS, etc. We implement an framework, BabelFish, to reuse these well-developed testing tools for x86 architecture. By using mature security tools we can give a more efficient and accuracy analysis. Meantime, our work can apply more complex analysis and optimizations without large overheads, unlike dynamic translation. In our approach, we leverage a translator to lift MIPS binary to IA-32 architecture. In order to guarantee the accurate testing results, a fine-translated code should be provided by the binary translator. This code is evaluated from two aspects, code correctness and runtime efficiency:

1) *Code Correctness*: The most important point is that the testing results provided by our framework should be consistent with those generated by testing tools working on MIPS binary directly. To ensure the code correctness, a good translator should translate 100% of the code if desired. Meanwhile, the translator should recover the correct control flow.

2) *Runtime Efficiency*: We would like to provide optimized translated code that can perform similar with the native code. Unlike existing static binary translator, we make an improvement on translating register operations to improve efficiency of translated code.

2 System Design

We design BabelFish, a framework to support security testing of MIPS binary (as presented in Figure 1). The input of our framework is a stripped MIPS binary (without any symbol information and debug information). BabelFish first translates the MIPS binary code to LLVM-IRs [1] statically.

Our approach provides two levels of security testing: the whole binary and specific functions in binary. Dealing with the whole binary, the translator will translate all necessary functions in MIPS binary. If we want to focus on testing some specified functions in MIPS binary, the translator of our framework also supports to translate part of functions in MIPS binary. The translator will only translate specified functions and all invoked functions according to call graph by static analysis,

The module of LLVM-IRs, outputted by the translator, consists of all translated functions and global array variables of relocated data. This module will be

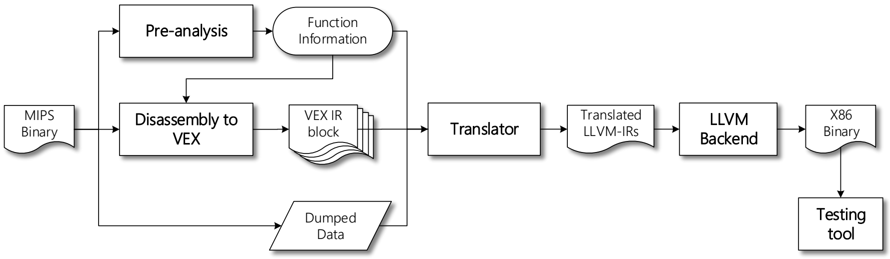

Fig. 1. Overview of BabelFish

compiled by LLVM backend into an executable binary or an object file. The x86 executable binary can be passed to testing tools of x86 binary for completing required security testing, like fuzzing, symbolic execution, taint analysis, etc. The object file can be linked into a test program for following security testing as well.

2.1 Challenge

Indirect Branch During binary translating, translating branch instructions, especially indirect branch instructions, is the key to preserve the accuracy of recovered control flow. Unlike direct branch, it is difficult to find the destination address of an indirect branch until runtime. The control flow targets are dynamically calculated based on some immidiate values in assembly instructions, jump tables and other referrences stored in data section. Combined with previous research [6], we first analyse the procedure of the indirect branch target calculation, and give a much accurate set of branch targets. With the set of branch targets, BabelFish implements indirect branch using the LLVM *switch* instruction, depending on a mapping relation between target addresses of the indirect branch and corresponding destination addresses in the translated code.

Data Relocation The data stored in data section is also an important portion in binary. The data will be dumped from data sections, and stored in arrays as global variables. While storing the data into the array, we must solve the relocatable problem. According to the research by Wang et al. [12], an immediate value can be a reference only if this value locates at the address space allocated for the binary. Here, We filter out references among all immediate values in data sections based on their target address. This simple filter is sufficient to identify concrete memory address. The immediate value as an operand in statements should be filtered as well. Once a reference is found during translating, it will be instantly replaced by the corresponding IR.

2.2 Binary Translating

The assembly-level binary translator first gives a pre-analysis on MIPS binary, utilizing an IDA script to export the information of function boundaries automatically. We also obtain the dynamic relocation entries of the file through objdump for dealing with the dynamic-link function calling.

With the pre-analysis results, we start to translate functions into LLVM-IRs individually. In order to facilitate translating a function call in LLVM-IR, we give a uniform form of translated function according to the most commonly used calling convention for 32-bit MIPS, the O32 ABI [2]:

Example of a Translated Function Described in C

```
uint32_t tranlated_func(const uint32_t args) {
// declare a array for the emulated stack frame
1:  uint8_t stack[64];
// load arguments storing in array args
2:  uint32_t arg0 = *(uint32_t *)args;
3:  uint32_t arg1 = *(uint32_t *)(args + 4);
...
// store callee's arguments to the stack array
4:  *(uint32_t *)stack = call_arg0;
5:  *(uint32_t *)(stack + 4) = call_arg1;
...
// call callee_func with the base address of stack
// as its only parameter
6:  uint32_t callee_ret = callee_func((uint32_t)stack);
...
}
```

At the beginning of each translated function, an array is allocated to complete translating the operation on stack frame with the same length of stack frame, and the translator loads the arguments from the address of array as the translated function's parameter (line 2-3). The length of stack frame is recorded to distinguish operations on the arguments of target function in assembly code. Subsequently, the translator follow the BFS ordering of function's control flow graph, and translated instructions in each basic blocks. Assembly code sequences in a basic block are firstly translated into VEX-IR representation, and then VEX-IR statements are translated into LLVM-IRs. VEX-IR abstracts binary code into a representation in a unified way, and lists all assembly side-effects, which allows for syntax-directed analysis.

During translating, we define the data type as *integer* or *pointer* type. What's more, the pointer variable is only used when we create it or we need to load/store the value from it, by leveraging the *ptrtoint* and *inttoptr* instruction in LLVM-IR. Once the translator get a pointer variable (e.g., like a memory reference or a return value of some system call), it convert its type to the integer without change its value using *ptrtoint* instruction. When we need to do load/store operation, we transfer the integer to a pointer, and then load the data that the pointer refers to, or store the data in where the pointer refers to. Havinging such transformation makes it convenient to do the other instruction translations, for their operands are always integer ones.

BabelFish sets shadow registers and shadow stack memory for each block to record the data's IR representation, data type, and the data value if it is a immediate value. Data information is real-timely updated during translating. The value stored in register may succeeds from previous blocks. Especially when a register succeeds from two or more blocks, we need to add a *phi* instruction to define its value at the beginning of this block. The *phi* instruction takes a list of [⟨*value*⟩, ⟨*block_label*⟩] pairs as arguments, which is based on the data flow.

3 Evaluation

We evaluate BabelFish with respect to its efficiency, the performance of translated code, and its capability to support program testing.

3.1 Experiments Setup

We tested BabelFish with code fragments about typical data transformation algorithms including cryptographic algorithm (AES, DES, RC4), hash algorithm (MD5,SHA1), compress algorithm (Huffman) and sort algorithm (Quick Sort, Bubble Sort). Besides, to understand the ability of translating a complete executable, we study our translators performance on *gzip* in detail. All programs to be translated are compiled by GCC 4.6.3 for MIPS32 architecture in little-endian, with the default configuration and optimization level -O2. We successfully translated all listed functions to LLVM-IRs. Then we lift translated IR to x86 assembly code using Clang 3.4. Finally, the translated x86 code is compared with the natively-compiled x86 executable binary, compiled by Clang 3.4 with -O2 optimization level, for the performance evaluation. The experiments are conducted on a machine with Intel Core i5-2320 @3GHz running Ubuntu 14.04.

3.2 Translating Efficiency

Here, we only consider the time consumed by BabelFish to translate MIPS code to LLVM-IRs. Processing time for each binary code is presented in Table 1. As expected, it takes more time to process larger functions. On average, BabelFish spends 0.137 seconds per function.

For *gzip*, There are 19K instructions and totally 154 functions, and it takes nearly 25 seconds to translate the whole MIPS binary to LLVM-IRs.

3.3 Translating Quality

The quality of translated code generated by BabelFish is evaluated from two aspects: efficiency and correctness.

Table 1. Characteristics of translating functions

Algorithm	**#Functions**	**#MIPS Insts**	**#IR Insts**	**translation time(s)**	**x86 Insts Expansion**
AES	4	1370	2492	0.707	17%
DES	2	2059	2775	0.653	4%
RC4	2	585	1435	0.375	64%
MD5	3	1198	1511	0.465	9.8%
SHA1	3	2063	2384	0.626	-12%
Huffman Coding	4	191	392	0.197	15%
Quick Sort	2	62	159	0.084	11%
Bubble Sort	1	22	66	0.044	-19%
total	21	7550	11214	3.154	7%

Correctness We verify the correctness of BabelFish by executing x86 binaries involving the translated functions with test input to verify the functionality. Semantic preservation is of significance to validity of testing the translated binary. We use the test cases provided by OpenSSL to check functionality of cryptographic functions and hash functions. As for Huffman coding and sort functions, we develop input by ourselves to verify the major functionality. All testing binaries pass the functionality tests without any error output.

Efficiency The efficiency of translated code manifests from two aspect: size expansion and the execution time.

Comparing the number of IR instructions with that of MIPS instructions, the average expansion is 48% (shown in Table 1), considering with extra instructions to convert temporary variable between pointer type and integer type as well as the *phi* instructions. However, the size of x86 assembly code compiled from translated IR is only 7% bigger than that native-compiled version.

Next, we examine the execution time of the translated code. We conduct the executing of source MIPS binaries on QEMU [4] and the translated code on host x86 machine directly. Figure 2 shows the normalized execution time of testing source MIPS binary and x86 test binary compared to its corresponding native-compiled version. Test binaries on x86 have almost the same execution time with the corresponding version compiled from source code. The translated code is of runtime efficiency.

Similarly, we compare the translated *gzip* binary executable with the native-compiled version. On one hand, the translated binary is 2.7 times larger than the native-compiled one, while the *.text* section of the translated binary is 1.5 times the size of that in the native-compiled version. Consider that the data in *.bss* and *.sbss* sections were dumped from source binary and then were initialized to 0 during translation, whose size is around 322 kilobytes. These data are stored in *.data* section of output binary. As *.bss* and *.sbss* sections are not calculated into the binary file size, the file expansion is acceptable. On the other hand, the translated executable roughly imposes 4% performance overhead compared with natively-compiled executable. The result shows that without any source code information, we can achieve a translated CPU intensive program using our framework, and such compiled binary have almost the same execution performance with the native-compiled version. This capability is convenient for testing a MIPS program by lifting it to an x86 version.

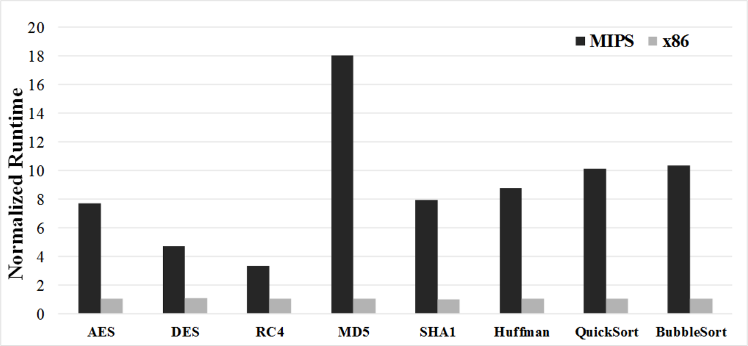

Fig. 2. Normalized execution time of source MIPS binary on QEMU and x86 translated binary compared to their natively-compiled version.

4 Related Work

Most security analysis and testing tools, mainly used for binary instrumentation, rewriting, and debugging, are based on same-ISA translators. Avater [13] is a framework to support dynamic security analysis of embedded devices firmwares based on S2E [5], and it orchestrates the communication between an emulator and a target physical device. PROSPECT [8] can provide an arbitrary analysis environments, and enable dynamic code analysis of embedded binary code inside the environments.

A static translator translates programs offline and can apply more extensive (and potentially whole program) optimizations. Bansal et al. [3] propose an efficient binary translation approach using superoptimization techniques. DisIRer [7] uses machine descriptions of GCC in reverse to translate 86 instruction sequences into GCCs low-level Register Transfer Language (RTL). LLBT [11] is the effort relied on the LLVM infrastructure as well, but it translates ARM binaries into LLVM IRs.

5 Conclusion

Since the lack of security testing tool in MIPS and the inconvenience of dynamic emulation, we want to make use of the well-designed tools for x86 executable.

Therefore, we propose BabelFish, a framework that translates the input MIPS binary code to LLVM-IR statically, then uses LLVM compiler mapping the IR code into IA-32. Subsequently, we can make all possible security testings of it. We have developed a prototype version and evaluated it with several MIPS binaries. Our experiments show that the translated binary code almost have the same performance with those compiled from source code. The quality of translated code is convenient for security testing.

References

1. The llvm compiler infrastructure. http://www.lllvm.org.
2. *MIPS32 ARCHITECTURE FOR PROGRAMMERS VOLUME II: THE MIPS32 INSTRUCTION SET.* 2001.
3. S. Bansal and A. Aiken. Binary translation using peephole superoptimizers. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 177–192. USENIX Association, 2008.
4. F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.
5. V. Chipounov, V. Kuznetsov, and G. Candea. The s2e platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)*, 30(1):2, 2012.
6. Y. Fu, Z. Lin, and D. Brumley. Automatically deriving pointer reference expressions from binary code for memory dump analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 614–624. ACM, 2015.
7. Y.-S. Hwang, T.-Y. Lin, and R.-G. Chang. Disirer: Converting a retargetable compiler into a multiplatform binary translator. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(4):18, 2010.
8. M. Kammerstetter, C. Platzer, and W. Kastner. Prospect: peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 329–340. ACM, 2014.
9. S. Parameswaran and T. Wolf. Embedded systems securityan overview. *Design Automation for Embedded Systems*, 12(3):173–183, 2008.
10. D. N. Serpanos and A. G. Voyiatzis. Security challenges in embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(1s):66, 2013.
11. B.-Y. Shen, J.-Y. Chen, W.-C. Hsu, and W. Yang. Llbt: an llvm-based static binary translator. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, pages 51–60. ACM, 2012.
12. S. Wang, P. Wang, and D. Wu. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, 2015.
13. J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, 2014.