Chapter 17

AUTOMATED VULNERABILITY DETECTION IN EMBEDDED DEVICES

Danjun Liu, Yong Tang, Baosheng Wang, Wei Xie and Bo Yu

Abstract Embedded devices are widely used today and are rapidly being incorporated in the Internet of Things that will permeate every aspect of society. However, embedded devices have vulnerabilities such as buffer overflows, command injections and backdoors that are often undocumented. Malicious entities who discover these vulnerabilities could exploit them to gain control of embedded devices and conduct a variety of criminal activities.

Due to the large number of embedded devices, non-standard codebases and complex control flows, it is extremely difficult to discover vulnerabilities using manual techniques. Current automated vulnerability detection tools typically use static analysis, but the detection accuracy is not high. Some tools employ code execution; however, this approach is inefficient, detects limited types of vulnerabilities and is restricted to specific architectures. Other tools use symbolic execution, but the level of automation is not high and the types of vulnerabilities they uncover are limited. This chapter evaluates several advanced vulnerability detection techniques used by current tools, especially those involving automated program analysis. These techniques are leveraged in a new automated vulnerability detection methodology for embedded devices.

Keywords: Embedded devices, automated vulnerability detection, binary analysis

1. Introduction

As the Internet of Things becomes more popular, massive numbers of embedded devices will permeate our lives. These devices include smart locks, smart utility meters, network routers, printers, surveillance systems, smart TVs, medical implants and automobiles. The tremendous growth of embedded devices has been spurred by advances in telecommunications and microelectronics. As far back as 2012, the Wi-Fi penetra-

G. Peterson and S. Shenoi (Eds.): Advances in Digital Forensics XIV, IFIP AICT 532, pp. 313–329, 2018.
https://doi.org/10.1007/978-3-319-99277-8_17

tion in South Korea reached 80% and nearly two-thirds of U.S. families had network routers; by 2020, the smart meter coverage in the United States will be close to 100% [16].

Firmware is the software that runs on embedded devices. It initiates the operating system, performs computations and realizes input/output operations. Just like general software, firmware may have vulnerabilities whose consequences can be very serious. For example, a home router is the only line of defense between a user's networked device and the Internet. Unfortunately, most vendors do not enhance home router security via updates. Therefore, once a router vulnerability is known, a malicious entity could, for example, modify router configurations and hijack network traffic [13, 18].

Embedded device analysis typically focuses on a specific version of firmware [10, 15, 22]. Although manual analysis can identify vulnerabilities in the firmware of an embedded device, it is a long and tedious process. Manual analysis does not scale – in the real-world, there are thousands of firmware images to be analyzed, and their underlying architectures, instruction sets and execution environments are diverse and complicated.

For these reasons, it is prudent to focus on the development of automated vulnerability detection techniques. Costin et al. [8, 9] have proposed automated analysis techniques, but the techniques have limited levels of automation and are unable to identify vulnerabilities in unknown firmware. Other researchers [22, 24] have also proposed automated firmware analysis techniques, but the automation is limited and only specific types of firmware vulnerabilities can be discovered.

Fortunately, several automated analysis tools have been developed for general programs based on source code, binary files, static analysis or dynamic execution. The techniques underlying these tools can be engaged in automating vulnerability detection in embedded devices. This chapter analyzes and evaluates the principal automated analysis techniques and leverages them in a new automated vulnerability detection methodology for embedded devices.

2. Vulnerability Detection Techniques

Several techniques have been proposed for program vulnerability detection. Some techniques have attracted considerable attention due to their good test results. However, even techniques that produce ordinary results have excellent underlying concepts that can be employed in developing enhanced techniques. This section discusses influential vulnerability detection techniques. In order to provide clarity, the techniques

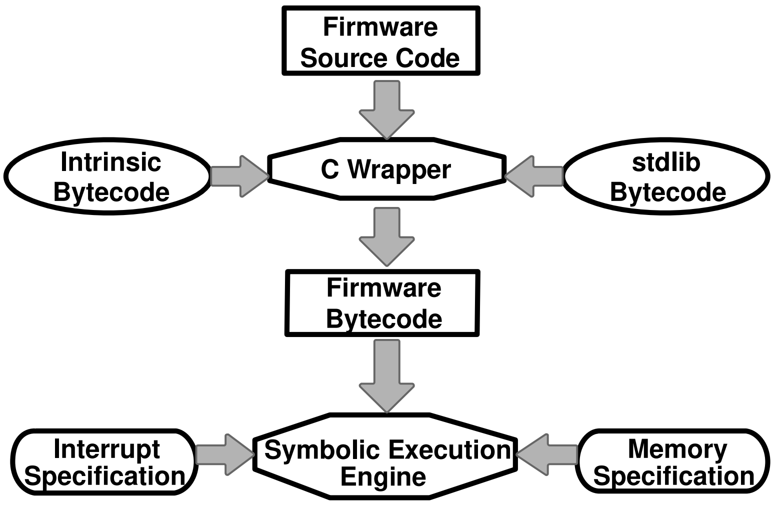

Figure 1. FIE workflow.

are divided into four categories: (i) source-code-based techniques; (ii) static-analysis-based techniques; (iii) actual-execution-based techniques; and (iv) symbolic-execution-based techniques.

2.1 Source-Code-Based Techniques

The FIE platform [11] is designed to detect vulnerabilities in firmware running on the MSP430 family of microcontrollers, primarily focusing on memory safety issues. Although FIE supports a specific architecture and relies on source code written in C, it provides a useful and effective framework for analyzing programs.

Figure 1 presents the FIE workflow. The source code of a target program is translated to LLVM bytecode and symbolic execution based on KLEE [3] is employed to analyze the bytecode. The symbolic engine supports 16-bit architectures and has a special memory structure that makes it easy to access program memory and hardware. It allows interrupts at any program statement and at any time during program execution. FIE performs an analysis of the entire firmware, but the symbolic execution can suffer from path explosion for certain loops, resulting in non-termination. However, this problem can be mitigated by state pruning, removing redundant (equivalent) states from the list of states to explore and performing "memory smudging" to concretize variables with respect to a finite set of values.

From the perspective of automating vulnerability discovery, FIE incorporates two key features. First, source code is translated to LLVM bytecode for symbolic execution. Second, effective techniques are applied to mitigate path explosion.

2.2 Static-Analysis-Based Techniques

Most static analysis techniques are used to supplement dynamic execution analysis; this is because they lack semantic understanding of programs. Specifically, these techniques are used to reason about programs without executing them, helping make proactive preparations for dynamic execution analysis.

Static analysis can provide assistance in understanding programs. The creation (i.e., recovery) of a control-flow graph (CFG) of a program, in which the nodes are basic blocks of instructions and the edges are possible control flow transfers between them, is a prerequisite for practically all static vulnerability discovery techniques. Control-flow graph recovery techniques are discussed in [17, 23, 28].

Another approach relies on the fact that certain kinds of program vulnerabilities have regular patterns. By analyzing graphs that embody program properties (e.g., control-flow graphs, data-flow graphs and control-dependence graphs), it is possible to construct vulnerability models that comprise sets of nodes in the graphs. Thus, finding program vulnerabilities is transformed to the problem of identifying vulnerability models in a program graph [19].

Yet another static analysis technique is value-set analysis [1]. This technique creates a value-flow graph (VFG), which provides an understanding of the possible targets of indirect jumps and memory write operations. It can detect buffer overflow vulnerabilities by analyzing overlapping buffers.

Costin et al. [8] have developed a framework that performs firmware collection, filtering, unpacking and analysis at a large scale (more than 30,000 firmware samples). They also proposed a correlation technique that propagates vulnerability information to similar firmware images. They discovered that one type of vulnerability affected 693 firmware images and reported 38 new common vulnerabilities and exposures (CVEs). Additionally, they searched for private encryption keys and known-bad strings to discover backdoors in devices whose firmware shared the same codebase with devices with known backdoors.

Wysopal et al. [27] have proposed a pattern-based, static analysis approach for detecting software backdoors. However, their approach requires the potential backdoors to be specified in advance. The corresponding patterns are used to perform the analysis. However, the search for pivotal strings needs to be improved. For example, a static ASCII string is more likely to represent a static password when it is referenced by or is located near a function that is known to be involved in the authentication process of an application.

2.3 Actual-Execution-Based Techniques

Fuzzing is the most practical vulnerability detection technique. It involves the creation and injection of mutated inputs to crash a target program.

Coverage-based fuzzing produces inputs that maximize the amount of code executed in a target application based on the insight that the greater the amount of code executed, the greater the possibility of executing vulnerable code. American Fuzzy Lop [30] is a security-oriented fuzzer that employs novel compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in a targeted binary. This substantially improves the functional coverage of the fuzzed code. The compact synthesized corpora produced by the tool are also useful for seeding other more labor- or resource-intensive testing regimes.

The second type of fuzzing is taint-based fuzzing [2]. It enhances the semantic understanding of target programs. Specifically, it helps understand which portions of the input must be mutated to execute new portion of the code.

Other techniques use actual execution to find specific types of vulnerabilities. An interesting example is PinTools, whose source code is available at GitHub (`github.com/JonathanSalwan/PinTools`). It performs actual execution using Pin, a dynamic binary instrumentation framework for the IA-32, x86-64 and MIC instruction-set architectures that enables the creation of dynamic program analysis tools [21]. PinTools uses behavior pattern matching to detect stack and heap overflows. It checks load and store instructions to determine the instructions that are outside the permitted area.

Schuster et al. [21] have proposed an approach for automatically identifying software backdoors in binary applications running on x86, x64 and MIPS32 architectures. They identify the specific regions in a binary that are prone to attacks and leverage this knowledge to determine suspicious components in an application.

Zaddach et al. [29] have developed the Avatar framework, which serves as an orchestration engine between a physical device and an external emulator based on S2E [27]. By injecting a special software proxy in the embedded device under test, Avatar can execute the firmware instructions in the emulator while channeling I/O operations to the physical

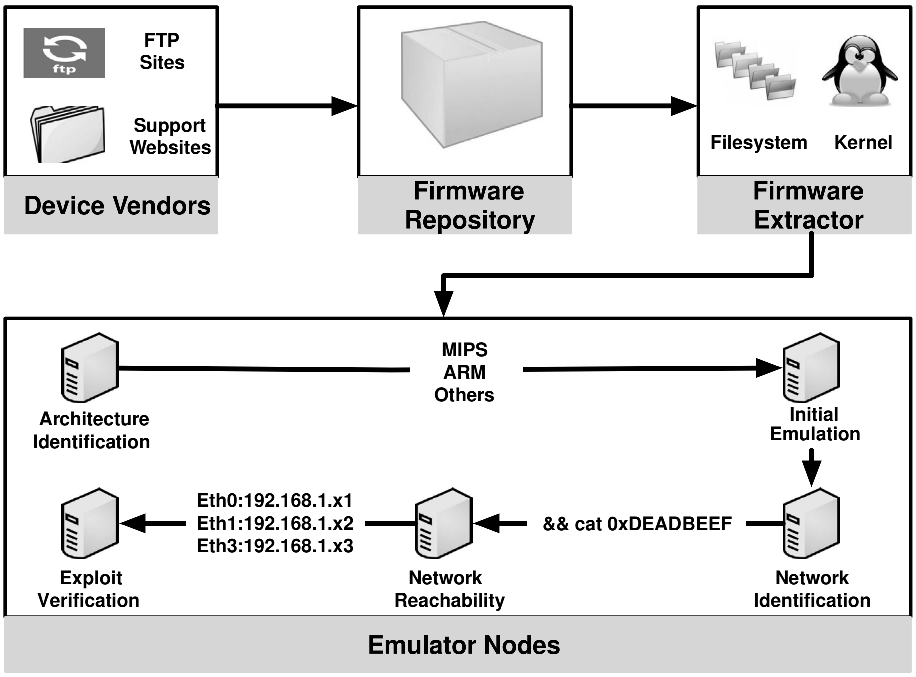

Figure 2. FIRMADYNE workflow.

hardware. The Avatar framework is able to detect vulnerabilities and backdoors.

FIRMADYNE [6] employs the most typical form of actual code execution. The tool uses an automated and scalable dynamic analysis technique to accurately identify vulnerabilities in Linux-based embedded firmware. Figure 2 presents the FIRMADYNE workflow. The tool automatically downloads firmware images from vendor websites. Next, it uses a custom extraction utility built around the `binwalk` API to extract the kernel (optional) and root filesystem contained in a firmware image. After it identities the architecture, the environment is dynamically reconfigured to match the expectations of the firmware image. Finally, the tool performs dynamic analysis.

2.4 Symbolic-Execution-Based Techniques

Symbolic execution techniques are generally used to determine the inputs that cause each portion of a program to execute. An interpreter follows the program, assuming symbolic values for inputs instead of obtaining actual inputs as a program would during normal execution; this

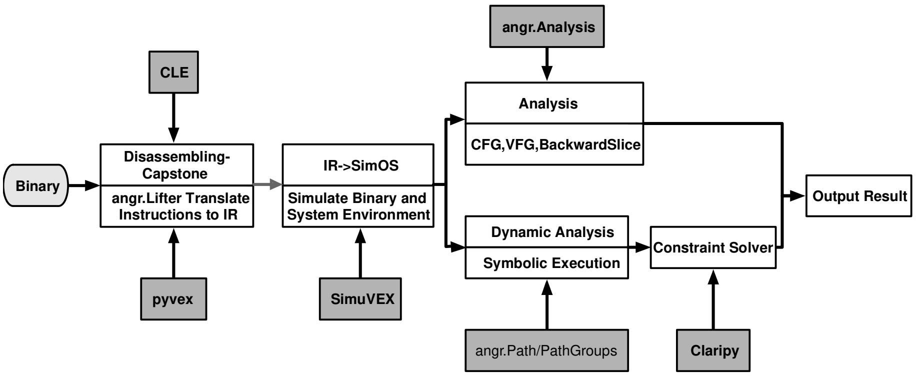

Figure 3. `angr` workflow.

approach is referred to as abstract interpretation. The interpreter produces expressions in terms of the symbolic values of program expressions and variables, and the possible outcomes of each conditional branch.

The most complete and powerful symbolic execution platform is Mayhem [5]. Its hybrid symbolic execution technique supports execution alternatives between online and offline symbolic execution runs; this speeds up the execution of multiple paths. Index-based memory modeling helps deal with code blocks in binaries.

Traditional dynamic symbolic execution requires considerable time, and loops and iterative constructs induce path explosion. To address these problems, Stephens et al. [26] developed the Driller tool, which leverages selective symbolic execution when performing fuzzing [4, 14]. The fuzzer is employed to explore the initial compartments of a program. When the fuzzer cannot go any further, concolic (i.e., concrete-symbolic) execution is employed to explore new compartments. Concolic execution, which involves under-constrained symbolic execution [20], can handle parts of programs without any function context.

The `angr` tool [25] implements all of the above symbolic execution techniques – binary and system state simulation, static analysis, dynamic analysis and constraint solving. Figure 3 presents the `angr` workflow. The tool performs the following steps:

- **Step 1:** The CLE module loads everything. The input binary is loaded, disassembled and converted to an intermediate representation IR [12].

- **Step 2:** Based on the generated intermediate language basic block, the binary with the system environment parameters is simulated and the corresponding class and data structures are generated to prepare for static analysis.

- **Step 3:** Static analysis is performed based on the available information, including from the control-flow graph, value-flow graph (VFG) and backward slice analysis.

- **Step 4:** Dynamic analysis is performed based on the available information (primarily symbolic execution), and various exploration strategies and debugging functions are applied.

- **Step 5:** The symbol execution constraints are solved.

- **Step 6:** The results are output.

3. Evaluation of Techniques

Vulnerability detection based on source code has a high degree of automation and good detection speed. Because the source code is available (instead of assembly code corresponding to a specific architecture), the technique has low dependence on architecture. The vulnerability detection efficiency is high, but its accuracy is average at best.

In order to enhance vulnerability detection, source code may be translated to an intermediate representation, whose execution is then simulated. The major limitation is the reliance on source code, which is often not available in the case of embedded devices; vendors typically provide compiled programs and updated firmware. Another limitation is that source code may not have vulnerabilities and vulnerabilities could be introduced during compilation. This also means that the same source code could manifest different vulnerabilities in different architectures. Additionally, even if vulnerabilities are found in source code, the compiler may add mechanisms that eliminate or exacerbate the vulnerabilities in the compiled code.

Vulnerability detection based on static analysis has a high degree of automation and the detection speed depends on the technique. This method can detect known types of vulnerabilities according to the established models. The goal is to derive a grammatical or semantic description of a vulnerability via grammatical or semantic level analysis of the program source code or binary. Static analysis is better suited to determining that a program does not contain certain errors.

A limitation is that static analysis cannot provide accurate run-time information. Additionally, if a target binary contains infeasible paths, false positives and omissions may result. Static analysis can find suspicious vulnerabilities, but it is difficult to reproduce an input that causes a crash; also, the replayability is poor. The most significant limitation is that static analysis is semantically poor. Although static analysis can tell what a program is, it cannot tell what the program is doing. This is why static analysis is often used to support dynamic execution.

Vulnerability detection based on actual execution has a low degree of automation and low detection speed. The detection efficiency is relatively low, but some unknown vulnerabilities can be discovered based on crash behavior. Dynamic analysis executes a test program to determine its run-time semantics. Dynamic information is accurate due to actual code execution, so there are no false positives. However, some omissions may occur because it is difficult to fully enumerate all the execution paths. The major difficulty with actual program execution is the proper configuration of the operating environment. As mentioned above, embedded devices have diverse architectures and complex library environments; this significantly complicates the task of automating the configuration process. For example, the FIRMADYNE tool [6] was able to simulate 96.6% of the firmware images it collected, but the network configuration of only 32.3% of the images could be inferred successfully. This causes the tool to be very unstable.

Vulnerability detection based on symbolic execution has a higher degree of automation than actual execution. The vulnerability detection efficiency is low, but unknown vulnerabilities can be discovered. The detection ability depends entirely on the prevailing constraints and the power of the constraint solver. Although the detection time is longer that for other methods, the detection accuracy is higher. Many wispy logic program errors can be identified, rendering symbolic execution appropriate for detecting specific problems. Branch statements are problematic because the number of paths grows exponentially relative to the size of the program. Additionally, loops and recursive structures can cause the number of paths to increase without bound. Indeed, path explosion is the major limitation of symbolic execution. Nevertheless, symbolic execution is attractive because assembly code for various architectures can be converted to a common intermediate code, following which the memory and running processes can be fully simulated; thus, multiple architectures are supported without the problems imposed by environmental configuration.

Table 1 summarizes the comparison results for the four program analysis techniques.

Table 1. Comparison of program analysis techniques.

	Source Code Analysis	**Static Analysis**	**Actual Execution**	**Symbolic Execution**
Vulnerabilities	Specific	Specific	Any	Anye
Replayable	No	No	Yes	Yes
Semantic Insight	Low	Low	High	High
Automation	High	High	Low	High
Architecture	Any	Specific	Partial	Any
Speed	Fast	Fast/Slow	Slow	Slow
Communication with Environment	No	No	Yes	No

4. Proposed Methodology

This section proposes a more comprehensive and efficient solution for detecting vulnerabilities in embedded devices. The proposed methodology leverages the advantages of the four program analysis techniques discussed above.

4.1 Design

Vendors typically do not release source code for their embedded devices; instead, they provide packaged firmware. This means that vulnerability detection methods based on source code are inappropriate. Additionally, as discussed above, there are numerous embedded devices with different architectures, reference libraries and operating environments. Embedded devices also differ in their endianness. As a result, environment configuration is a big problem. Since large-scale, automated analysis is desired, vulnerability detection techniques based on actual execution are inappropriate.

All things considered, an ideal methodology should apply static analysis followed by symbolic execution to support large-scale, automatic analysis. Thus, the proposed detection methodology is based on the `angr` framework. The analysis target is based on FIRMADYNE [6], which collects and unpacks a number of firmware images.

Figure 4 shows the workflow of the proposed vulnerability detection methodology.

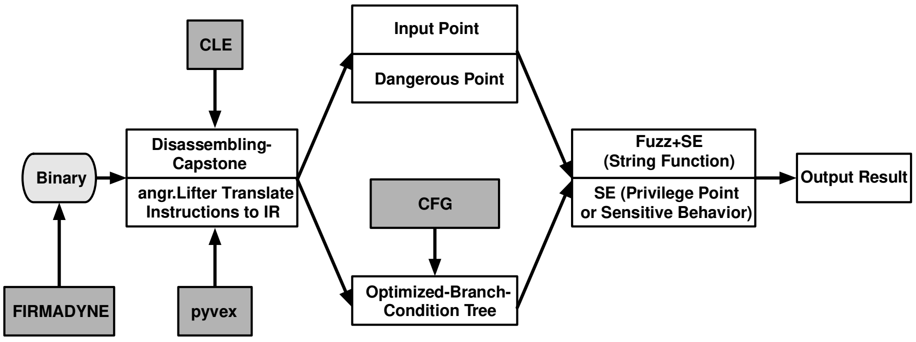
CLE
Input Point
Dangerous Point
Binary
Disassembling-Capstone
angr.Lifter Translate Instructions to IR
CFG
Fuzz+SE (String Function)
SE (Privilege Point or Sensitive Behavior)
Output Result
FIRMADYNE
pyvex
Optimized-Branch-Condition Tree

Figure 4. Proposed vulnerability detection methodology workflow.

4.2 Static Analysis Stage

The static analysis stage involves two main tasks that automatically identify the binary input points and program execution points that are likely to induce vulnerabilities.

Table 2. Input points.

Type	**Function**
Command Line Arguments	`argv` operation
Environment Variable	`getenv()`
File Data	`read()`, `fscanf()`, `getc()`, `fgetc()`, `fgets()`, `vfscanf()`
Keyboard Input/stdin	`read()`, `scanf()`, `getchar()`, `gets()`
Network Data	`read()`, `recv()`, `recvfrom()`

- **Input Points:** The library functions in Table 2 may be invoked to obtain inputs. Due to the specificity of embedded devices, their binaries rarely receive inputs from such obvious library functions. However, the common gateway interface (CGI) for embedded devices can parse user requests and send the parsed requests to other binaries. This method is used by the proposed methodology to provide inputs to a target binary.

  Real embedded devices have complex firmware, which makes it very difficult to identify input points. One approach is to search for the memory addresses of input values based on the parameter-passing convention used by a function call. Additionally, there is

Table 3. Dangerous points.

Dangerous Point Type	**Vulnerability Type**	**Dangerous Points**
String-Related Function	Buffer Overflow	String Copying: `strcpy()`, `strncpy()`; String Combination: `strcat()`; String Formatting: `sprintf()`, `snprintf()`
Privileged Program Points	Backdoor, Command Injection	Command Execution: `system()`, `execve()`, `setuid()`, `setgid()`
Sensitive Behavior	Backdoor, Command Injection	Peripheral Device Access; Suspicious String Output; Sensitive Memory Access

always a binary program in firmware – called the initial binary – that calls a library function to handle a user request. The initial binary acts as the input point to multiple binaries, enabling all the binaries to be analyzed at the same time.

- **Dangerous Points:** A dangerous point is a program execution point that could cause a vulnerability. A dangerous point can be a specific statement that a program executes or it can be a certain type of behavior exhibited by the program. Table 3 provides information about the three types of dangerous points.

4.3 Execution Preparation Stage

The execution preparation stage uses `pyvex` to translate assembly code for different architectures to the intermediate language IR, which is used to simulate execution. This feature supports multi-architecture vulnerability detection. During this stage, a control-flow graph created by `angr` is employed to construct the optimized-branch-condition tree. The condition tree expresses the constraint that an input should satisfy at an arbitrary point in the program; therefore, it contains the input constraint to a dangerous point in the program. The conditional tree is used in subsequent executions.

Unlike the condition tree employed by `angr`, the optimized-branch-condition tree used in the proposed methodology assumes that every part of the input value has no multiple or crossed meanings. Thus, before the execution flow reaches a point, there is no effect on the program if the input condition changes its evaluation order. For exam-

```
 1   int main(void) {
 2       char user_command[10];
 3       int user_hash;
 4
 5       read(0, user_command, 10);
 6       read(0, user_hash, sizeof(int));
 7
 8       if (user_hash != hash(user_command)) {
 9           puts("Hash mismatch!");
10           return 1;
11       }
12
13       if (strncmp("CRASH", user_command, 5) == 0) {
14           puts ("Welcome to compartment 3!");
15           if (user_command[5] == '!') {
16               path_explosion_function();
17               if (user_command[6] == '!') {
18                   puts ("CRASHING!");
19                   abort();
20               }
21           }
22       }
23
24       return 0;
25   }
```

Figure 5. Code example from Driller [26].

ple, a user may submit a request using a URL with format `http://[router-address]/cgi-bin/;uname\$IFS-a`. Each word has a unique meaning and mutual evaluation has no effects. Therefore, when constructing the condition tree, the order of conditional evaluation can be changed appropriately.

The optimized-branch-condition tree facilitates the understanding of complex conditional evaluations. Figure 5 shows a code example from Driller [26]. The program has a generic input and a specific input at the same time. A generic input has a wide range of valid values (e.g., user name) whereas a specific input has a limited set of valid values (e.g., name hash). In the program, the `user_command` parameter is treated as a generic input (Line 8) and as a specific input (Line 13), causing Driller to switch between the fuzzer and concolic execution engine. At this point, Driller gets stuck. The optimized-branch-condition tree can change the order of evaluation of `user_command`, preventing the analysis from getting stuck.

4.4 Symbolic Execution Stage

The performance of the symbolic execution step depends on the type of vulnerability to be detected. For example, if the dangerous point is a function involving string manipulation (i.e., where a buffer overflow may occur), then a smart fuzzing method is applied. The fuzzer mutates the inputs in an intelligent manner based on the input constraints for the string manipulation function, where the constraints are embodied in the optimized-branch-condition tree.

Next, mutated values are used as input values and the execution of the program is simulated. During the execution, crashes are recorded. Moreover, the behavior of the program is monitored to check if the string manipulation function goes out of bounds or if load and/or store instructions in an assignment loop are out of bounds. A program crash or cross-border operation may indicate a buffer overflow vulnerability.

The simulation method may be applied directly if the dangerous point is a privileged program point or exhibits sensitive behavior, where a backdoor or command injection vulnerability may occur. The actual input value of the target program is replaced with a symbolic value that satisfies the input constraints to a privileged program point, following which the simulation is executed. A backdoor or command injection vulnerability may exist if the program can reach the privilege point or generate sensitive behavior.

Optimizations should be performed during concolic execution to avoid path explosion. In this context, the `angr` tool incorporates two interesting strategies, function summarization and path exploration. Function summarization replaces irrelevant library function calls for which there is no need to detect internal errors. Path exploration is good for path pruning. For example, if execution falls into a loop, the loop is only executed three times. Other innovative approaches may also be applied as long as the final results are not affected.

5. Conclusions

Vulnerabilities in embedded devices can be exploited to conduct any number of criminal activities. However, due to the large number of embedded devices, non-standard codebases and complex control flows, it is extremely difficult to discover vulnerabilities using manual techniques. The evaluation of automated program analysis techniques – source-code-based analysis, static analysis, actual execution and symbolic execution – has revealed various advantages and limitations. The proposed methodology for vulnerability detection leverages the advantages of existing program analysis techniques. Nevertheless, numerous opportunities are available for enhancing the methodology by incorporating innovative strategies and tactics, including artificial intelligence and machine learning techniques.

References

[1] G. Balakrishnan and T. Reps, WYSINWYX: What you see is not what you execute, *ACM Transactions on Programming Languages and Systems*, vol. 32(6), article no. 23, 2010.

[2] S. Bekrar, C. Bekrar, R. Groz and L. Mounier, A taint-based approach for smart fuzzing, *Proceedings of the Fifth IEEE International Conference on Software Testing, Verification and Validation*, pp. 818–825, 2012.

[3] C. Cadar, D. Dunbar and D. Engler, KLEE: Unassisted and automatic generation of high-coverage tests for complex system programs, *Proceedings of the Eighth USENIX Conference on Operating Systems Design and Implementation*, pp. 209–224, 2008.

[4] D. Caselden, A. Bazhanyuk, M. Payer, L. Szekeres, S. McCamant and D. Song, Transformation-Aware Exploit Generation using a HI-CFG, Technical Report No. UCB/EECS-2013-85, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, California, 2013.

[5] S. Cha, T. Avgerinos, A. Rebert and D. Brumley, Unleashing Mayhem on binary code, *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 380–394, 2012.

[6] D. Chen, M. Egele, M. Woo and D. Brumley, Towards automated dynamic analysis for Linux-based embedded firmware, *Proceedings of the Twenty-Third Annual Network and Distributed System Security Symposium*, 2016.

[7] V. Chipounov, V. Kuznetsov and G. Candea, S2E: A platform for in-vivo multi-path analysis of software systems, *ACM SIGPLAN Notices*, vol. 46(3), pp. 265–278, 2011.

[8] A. Costin, J. Zaddach, A. Francillon and D. Balzarotti, A large-scale analysis of the security of embedded firmware, *Proceedings of the Twenty-Third USENIX Security Symposium*, pp. 95–110, 2014.

[9] A. Costin, A. Zarras and A. Francillon, Automated dynamic firmware analysis at scale: A case study on embedded web interfaces, *Proceedings of the Eleventh ACM Asia Conference on Computer and Communications Security*, pp. 437–448, 2016.

[10] A. Cui, M. Costello and S. Stolfo, When firmware modifications attack: A case study of embedded exploitation, *Proceedings of the Twentieth Annual Network and Distributed System Security Symposium*, 2013.

[11] D. Davidson, B. Moench, S. Jha and T. Ristenpart, FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution, *Proceedings of the Twenty-Second USENIX Security Symposium*, pp. 463–478, 2013.

[12] T. Dullien and S. Porst, REIL: A platform-independent intermediate representation of disassembled code for static code analysis, *Proceedings of the Ninth CanSecWest Applied Security Conference*, 2009

[13] A. Fournaris, L. Pocero Fraile and O. Koufopavlou, Exploiting hardware vulnerabilities to attack embedded system devices: A survey of potent microarchitectural attacks, *Electronics*, vol. 6(3), article no. 52, 2017.

[14] P. Godefroid, N. Klarlund and K. Sen, DART: Directed automated random testing, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 213–223, 2005.

[15] B. Gourdin, C. Soman, H. Bojinov and E. Bursztein, Toward secure embedded web interfaces, *Proceedings of the Twentieth USENIX Security Symposium*, 2011.

[16] M. Hachman, U.S. barely cracks list of countries with top Wi-Fi penetration, *PC Magazine*, April 5, 2012.

[17] J. Kinder and H. Veith, Jakstab: A static analysis platform for binaries, *Proceedings of the Twentieth International Conference on Computer Aided Verification*, pp. 423–427, 2008.

[18] D. Papp, Z. Ma and L. Buttyan, Embedded systems security: Threats, vulnerabilities and attack taxonomy, *Proceedings of the Thirteenth Annual Conference on Privacy, Security and Trust*, pp. 145–152, 2015.

[19] J. Pewny, B. Garmany, R. Gawlik, C. Rossow and T. Holz, Cross-architecture bug search in binary executables, *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 709–724, 2015.

[20] D. Ramos and D. Engler, Under-constrained symbolic execution: Correctness checking for real code, *Proceedings of the Twenty-Fourth USENIX Security Symposium*, pp. 49–64, 2015.

[21] J. Salwan, Stack and heap overflow detection at runtime via behavior analysis and Pin (`shell-storm.org/blog/Stack-and-heap-overflow-detection-at-runtime-via-behavior-analysis-and-PIN`), October 14, 2010.

[22] F. Schuster and T. Holz, Towards reducing the attack surface of software backdoors, *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 851–862, 2013.

[23] B. Schwarz, S. Debray and G. Andrews, Disassembly of executable code revisited, *Proceedings of the Ninth Working Conference on Reverse Engineering*, pp. 45–54, 2002.

[24] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel and G. Vigna, Firmalice – Automatic detection of authentication bypass vulnerabilities in binary firmware, *Proceedings of the Twenty-Second Annual Network and Distributed System Security Symposium*, 2015.

[25] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel and G. Vigna, SOK: (State of) The art of war: Offensive techniques in binary analysis, *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 138–157, 2016.

[26] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel and G. Vigna, Driller: Augmenting fuzzing through selective symbolic execution, *Proceedings of the Twenty-Third Annual Network and Distributed System Security Symposium*, 2016.

[27] C. Wysopal, C. Eng and T. Shields, Static detection of application backdoors, *Datenschutz und Datensicherheit*, vol. 34(3), pp. 149–155, 2010.

[28] L. Xu, F. Sun and Z. Su, Constructing Precise Control Flow Graphs from Binaries, Technical Report, Department of Computer Science, University of California, Davis, Davis, California, 2009.

[29] J. Zaddach, L. Bruno, A. Francillon and D. Balzarotti, Avatar: A framework to support dynamic security analysis of embedded system firmware, *Proceedings of the Twenty-First Annual Network and Distributed System Security Symposium*, 2014.

[30] M. Zalewski, American Fuzzy Lop (`lcamtuf.coredump.cx/afl`), 2017.