

Securing Operating Systems Through Fine-grained Kernel Access Limitation for IoT Systems

Dongyang Zhan, *Member, IEEE*, Zhaofeng Yu, Xiangzhan Yu, Hongli Zhang, Lin Ye, and Likun Liu*

Abstract—With the development of Internet of Things (IoT), it is gaining a lot of attention. It is important to secure the embedded systems with low overhead. The Linux Seccomp is widely used by developers to secure the kernels by blocking the access of unused syscalls, which introduces less overhead. However, there are no systematic Seccomp configuration approaches for IoT applications without the help of developers. In addition, the existing Seccomp configuration approaches are coarse-grained, which cannot analyze and limit the syscall arguments. In this paper, a novel static dependent syscall analysis approach for embedded applications is proposed, which can obtain all of the possible dependent syscalls and the corresponding arguments of the target applications. So, a fine-grained kernel access limitation can be performed for the IoT applications. To this end, the mappings between dynamic library APIs and syscalls according with their arguments are built, by analyzing the control flow graphs and the data dependency relationships of the dynamic libraries. To the best of our knowledge, this is the first work to generate the fine-grained Seccomp profile for embedded applications.

Index Terms—Shrinking attack surface, systematic static analysis, dependent variable analysis, IoT security.

I. INTRODUCTION

INTERNET of Things (IoT) is developing rapidly and has become one of the most important computing platforms. IoT devices have been used in a lot of environments, such as industrial control/sensing systems, home automation, etc. However, the security risks are raised with the IoT development. The IoT/embedded systems have strict requirements for resource occupation and energy consumption, so the security problem is more serious [1], [2]. And, the performance and resource requirements of security tools running inside the embedded devices are also very strict [3].

Securing embedded operating systems is essential for IoT security [4]. There are many vulnerabilities [5]–[7] in OS kernels that can be exploited to perform the privilege escalation attacks from applications. After the privilege escalation, attackers can control the device and access all of the data. Therefore, securing the embedded operating system and isolating the IoT applications are important.

Fortunately, the Linux Seccomp can be used to secure the operating systems. According to the Eclipse IoT Developer Survey report, Linux is the most popular OS for IoT devices [8]. Seccomp is a security module of the Linux kernels, which can block the unused syscalls from user-space applications.

Since the vulnerabilities in the blocked syscalls are isolated, the attack surface is reduced. Seccomp is an embedding module in the Linux kernel, which usually introduces low overhead to the whole system, so it is meaningful to secure the embedded systems based on Seccomp [9].

By using Seccomp, the kernel attack surface can be reduced by preventing the unused kernel syscalls that may contain exploitable vulnerabilities from being exploited by user-space applications. However, it is challenging to generate Seccomp configuration systematically. To analyze the dependent syscalls of a binary, the dynamic tracking and static analysis approaches are proposed for x86 binaries. The dynamic tracking approaches collect the invoked syscalls of the target binaries dynamically. But it cannot obtain the complete dependent syscall list, which is not acceptable in practical scenarios. The static analysis approaches generate the mapping between the dependent library APIs and syscalls, so that the syscall list can be obtained. But, these systems are not designed for embedded systems. Furthermore, the existing static and dynamic analysis approaches are coarse-grained, which cannot analyze the mapping between library API (e.g., `fopen`, `fclose`) and syscall arguments. Limiting syscall arguments is essential for system security, because some arguments affect the control flows and the functions of the syscalls.

In this paper, a systematic Seccomp configuration analysis approach is proposed to reduce the attack surface for embedded devices (e.g., routers), which can not only block the unused syscalls but also limit the arguments of the allowed syscalls. The dependent syscalls and the corresponding arguments are obtained by analyzing the target binaries and the dependent libraries. Firstly, the target binaries are analyzed to obtain the invoked library APIs and the corresponding arguments. Then, the mapping between library APIs and syscalls is built by analyzing the dependent libraries. To further limit the syscall arguments, the relationship between API arguments and syscall arguments is analyzed. Finally, the fine-grained Seccomp configuration can be generated.

To generate the mapping between library API arguments and syscall arguments, a systematic data dependency analysis approach is proposed, which can construct a data dependency graph for every library API argument. The graph is constructed by control flow graph analysis, static taint analysis and symbolic execution. Firstly, the control flow graph of every library API is constructed, so that the mapping between library APIs and syscalls can be obtained. Then, the static taint analysis is employed to track the propagation of every argument. Static taint analysis and dynamic data tracking are two kinds of data flow analysis methods. Since we aim to

D. Zhan, Z. Yu, X. Yu, H. Zhang, L. Ye, L. Liu are with the School of Cyberspace Science, Harbin Institute of Technology, Harbin, Heilongjiang, 150001.

E-mail: {zhandy, yuxiangzhan, zhanghongli, hityelin, liuliku}@hit.edu.cn

* Corresponding Author: liuliku@hit.edu.cn

find out the comprehensive data dependency relationship, we use the static approach in this paper. During the taint analysis and symbolic execution, the variables that influence the data propagation are analyzed to find out if the syscall arguments are determined based on the library API arguments. If so, the syscall arguments can be determined.

To the best of our knowledge, this is the first work to perform the fine-grained syscall access control for the embedded systems. By analyzing the target applications and the dependent libraries, the types and arguments of the system calls can be limited at best effort. Therefore, the embedded system can be secured with very low overhead by leveraging Seccomp.

In summary, the contributions of our paper are as follows.

- A systematic attack surface reduction approach based on Seccomp is proposed to secure the embedded systems by limiting the accessible syscalls and the corresponding arguments of the embedded applications.
- The applications and the dependent libraries are analyzed statically to find out the comprehensive dependent syscalls, so that the corresponding Seccomp configuration can be generated.
- To further reduce the attack surface, a data dependency analysis approach is proposed, which constructs the data dependency graph of every syscall argument to generate the mapping between API arguments and syscall arguments.

The rest of this paper is organized as follows. The background and key insight are described in Section II. Section III presents the system design. Section IV gives some implementation details of our system. Section V evaluates our system. The related work is summarized in Section VI. Finally, Section VII concludes this paper.

II. BACKGROUND & KEY INSIGHT

A. Linux Seccomp

There is a computer security facility in the Linux kernel called Seccomp (secure computing mode) [10], which allows processes to use Berkeley Packet Filter (BPF) to filter syscalls through configurable policies. The Linux kernel exposes many syscalls to user space. In recent years, many discovered vulnerabilities are related to Linux syscalls, and some of these vulnerabilities (e.g., CVE-2017-7308, CVE-2017-5123) may be used to damage the operating system kernel. It can be seen that the syscalls exposed to user space introduce a large attack surface. Considering that the user space process only uses a subset of the syscall set provided by the Linux, we can narrow the attack surface by reducing the set of syscalls accessible by the user-space processes. And, Seccomp provides a method for the processes to specify the set of syscalls they can call. Seccomp is typically configured in two ways. First, an application can configure the Seccomp policy by using several syscalls. Second, an application can first configure the Seccomp and then execute another application that should be protected, so the executed application will also be limited with the same Seccomp policy. When Seccomp is configured, if the process tries to access a syscall that is not allowed,

the operating system will reject the call. The Linux Seccomp is widely used by lots of well-known programs, such as OpenSSH, etc. However, many developers do not specifically provide Seccomp configuration files, so this may expose the operating system to certain risks. If an unrestricted program that provides Internet services is controlled by an attacker in some way, the vulnerabilities in all accessible syscalls may be exploited. Therefore, strengthening the security of the operating system through Seccomp is very important for the maintainer of the system. This paper aims to narrow the attack surface by precisely restricting the syscalls that the program can access without requiring additional work from the developers.

B. Shrinking Kernel Attack Surface

Shrinking the kernel attack surface is one of the most important approaches to secure the systems [11], based on the observation that the fewer system calls an application can access, the less chance it can exploit the kernel vulnerabilities [12]. There are some approaches to reduce the attack surface in different scenarios.

Recompiling the programs with the library OS [13] to reduce the dependent syscalls is a possible way. The Nabla Container [12] integrates most of the syscalls into the container images by redesigning the applications and recompiling them with the library OS. After that, only 7 syscalls can be accessed by the containers. The experimental results show that the accessed code of the tested Nabla Container is less than that of Docker containers. But, not all applications can be recompiled, so the idea of Nabla Container is not widely applied by industry.

Seccomp is widely used to reduce the available syscalls for applications. By using Seccomp flexibly, SPEAKER [14] can apply different control policies to an application according to its execution status (e.g., initialization and servicing). SPEAKER collects the dependent syscall sets by tracking the target applications dynamically. The dynamic syscall collection approaches are also employed by [15]–[17], which can generate Seccomp policies for applications based on the invoked syscalls. However, the dynamic syscall collection cannot obtain the comprehensive dependent syscalls because it is difficult for dynamic execution to cover all of the execution paths.

To overcome the problems of dynamic syscall collection, some static dependent syscall analysis approaches are proposed. Confine [11] is designed for containers, which builds the mapping between library APIs and syscalls based on the static analysis of the library source code. After obtaining the dependent library APIs, Confine can generate Seccomp configuration for containers. Unlike Confine, Sysfilter [18] and [19] builds the mapping between library APIs and syscalls based on the static analysis of library binaries. Chestnut [20] analyzes the dependent syscalls through a static compiler-based source code analyzer and a binary analyzer, and it can restrict the set of allowed syscalls dynamically. However, these approaches cannot analyze the arguments of the dependent syscalls, making the access control coarse-grained. In addition,

these approaches are designed for x86 programs, and no system is designed for embedded applications. There are several challenges to apply the approaches for x86 programs to ARM programs, due to some special designs for the ARM platform. First, the syscall mechanism of the ARM platform is different from that of x86. Second, the inline assembly code used to invoke syscalls in glibc under the ARM platform is different from that under the x86 platform, so the binary analysis should be redesigned for the ARM platform. Furthermore, the instruction analysis and the data propagation analysis of ARM applications are different from those of x86 applications. In this paper, we aim to propose a fine-grained syscall limitation approach to limit both syscalls and their parameters precisely for embedded IoT applications.

C. Threat Model

Our approach aims to reduce the kernel attack surface by preventing the unused kernel syscalls that may contain exploitable vulnerabilities from being exploited by user-space applications. We assume that the attacker can hijack the control flow of user-space applications (or call the target applications) to invoke some syscalls that contain vulnerabilities. Before the attacks, the kernel is secure and not controlled by attackers.

We assume the source code of popular dependent libraries is available, especially for glibc, which is open-source and usually used by programs or other dynamic libraries to invoke syscalls on different platforms (e.g., x86 and ARM). For the self-written libraries and the programs that use inline assembly code to invoke syscalls, we analyze the binaries of them. In order to increase the generality of our system, we analyze the binaries of the target programs.

D. Observation

For Linux-based systems, the dependent syscalls of a program can be obtained by analyzing the dependent libraries. The user-space programs usually invoke syscalls through dynamically-linked libraries. A program can rely on many libraries (e.g., libssl, glibc, etc.). Among them, the glibc is one of most important one, since it is usually used by programs or other libraries to invoke syscalls. In addition, most self-written libraries also leverage the glibc to invoke syscalls. Since glibc is open-source, we can obtain the source code of it. Based on the analysis of the code, we can build the mapping between the library APIs and syscalls. Based on the dependent API list of a target program, the dependent syscalls can be obtained.

To perform the dependent syscall analysis for IoT applications, the target binary should be firstly analyzed to obtain the dependent libraries and the corresponding APIs. For Linux, most of dependent libraries (e.g., glibc and other basic libraries) are open-source, so it is possible to analyze the mapping based on the open-source libraries. In some systems, some dependent libraries can even be replaced. There are two scenarios of applying the Seccomp-based security tools in embedded systems. When the application can be modified by the security administrators, the generated Seccomp configuration can be embedded in the application through the configuration syscalls. If the application cannot be modified, the application

can be executed by another programmable application and Seccomp is configured in the programmable application. So that, the application is also limited.

Seccomp is able to filter the syscall arguments (i.e., flags), which can further secure the system. Syscall arguments include many types, such as flags and pointers, and most of them are passed from the library API parameters. But, the syscall arguments that Seccomp can filter are flags. Seccomp cannot filter points. But, currently there is no automatic argument analysis tool to enrich Seccomp configuration with argument policies. Filtering syscall arguments is important for system security, because many attacks [21], [22] rely on some critical syscalls with special arguments. For instance, some backdoor attacks [22] tamper with the original login file of the target server. If the application cannot invoke the open syscall with the write flag, these attacks can be blocked.

Besides, limiting syscall arguments can reduce the number of accessible kernel functions of a syscall, which can reduce the possibility of triggering kernel vulnerabilities. We select several syscalls (e.g., socket, access, fchmod, etc.) to do the experiment by tracking the invoked kernel functions with different syscall arguments (or flags). In the experiment, we use trace-cmd to track and record the kernel functions accessed by each syscall given different arguments, which results are shown in Figure 1. The blue bar shows the number of executed kernel functions of a syscall with all possible arguments, and the red bar shows the minimal function number with a specific argument. From the figure of 9 examples, we can find that limiting the syscall arguments can significantly reduce the possible accessed kernel functions. Taking the socket syscall as an example, we take AF_INET, AF_INET6, and AF_UNIX for the first argument, and SOCK_STREAM, SOCK_DGRAM, and SOCK_RAW for the second argument. In this way, a total of nine argument usage situations are formed. As shown in Table I, when the value of the argument is not restricted, we tracked 299 executed kernel functions. When argument 1 is restricted to AF_INET, AF_INET6, and AF_UNIX, the number of executed kernel functions is 267, 252, and 214, respectively. Limiting only one argument can greatly reduce the number of kernel functions accessed by a syscall. When the remaining arguments are further restricted, we can further reduce the number of accessed kernel functions. From Table I, we can see that the results of mmap are similar with those of socket. Based on the assumption that the more accessible code, the greater the probability of exploiting the vulnerabilities [12], further limiting syscall arguments is essential for system security.

Fortunately, analyzing the possible dependent API arguments of programs can be performed by [23]. Based on the idea of it, it is possible to analyze the dependent API arguments of IoT applications. So, the focus of this paper is to further construct the mapping between API arguments and syscall arguments, so that Seccomp can be used to secure the embedded IoT system with low overhead.

TABLE I
NUMBERS OF EXECUTED KERNEL FUNCTIONS WITH DIFFERENT LIMITED SYSCALL ARGUMENTS.

syscall	Total Num	Arg1	Num2	Arg2	Num3
socket	299	AF_INET	267	SOCK_STREAM	211
				SOCK_DGRAM	234
				SOCK_RAW	252
		AF_INET6	252	SOCK_STREAM	221
				SOCK_DGRAM	196
				SOCK_RAW	227
		AF_UNIX	214	SOCK_STREAM	211
				SOCK_DGRAM	207
				SOCK_RAW	206
mmap	144	PROT_EXEC	120	MAP_SHARED	118
				MAP_PRIVATE	120
		PROT_READ	117	MAP_SHARED	114
				MAP_PRIVATE	117
		PROT_WRITE	138	MAP_SHARED	134
				MAP_PRIVATE	118
		PROT_NONE	116	MAP_SHARED	107
				MAP_PRIVATE	116

Total Num: the number of invoked kernel functions with all possible arguments; Arg1: the first argument of the syscall; Num2: the number of invoked kernel functions with the first argument fixed; Arg2: the second argument of the syscall; Num3: the number of invoked kernel functions with the first and second arguments fixed.

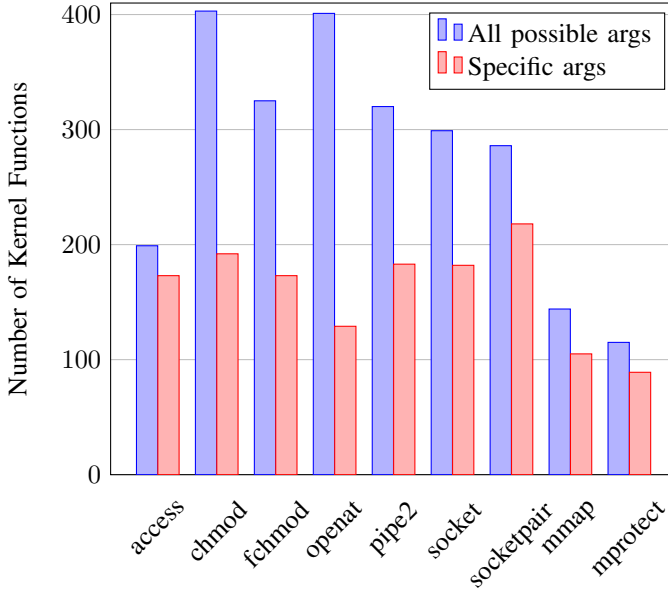


Fig. 1. Comparison of invoked kernel functions of syscalls with different arguments.

III. SYSTEM DESIGN

A. System Overview

Our system does not block a list of vulnerable syscalls/arguments, but only allows necessary syscalls/arguments for programs to execute. So that, the vulnerabilities caused by blocked syscalls/arguments can be mitigated. The reason we do not block only vulnerable syscalls/arguments is that the vulnerable syscalls we get may be incomplete. Our system identifies the list of necessary syscalls/arguments for a program to execute, and blocks other syscalls/arguments. So, the vulnerabilities caused by all of the unnecessary syscalls/arguments will be mitigated. As a result, our system minimizes the OS attack surface.

There are two main stages in our system: the mapping construction stage and the binary analysis stage, which workflow is shown in Figure 2. The mapping construction analyzes the dependent libraries to construct the mapping between library APIs and syscalls, which also includes the mapping between API arguments and syscall arguments. The binary analysis extracts the dependent library APIs and the directly invoked syscalls according with the arguments of APIs and syscalls. Finally, the fine-grained Seccomp configuration can be generated based on the mapping and the dependent APIs.

The mapping construction stage aims to build the mapping between library APIs and syscalls and the arguments of them. To that end, the control flow graph of the dependent library is first analyzed. By searching the control flow graph, we can find the related syscalls of every library API. The control flow graphs of dependent libraries are analyzed by two steps, including the direct call graph analysis and the indirect call graph analysis, which are inspired by Confine [11] and [24]. We design a compiler-based function call graph construction approach to construct the direct call graph. As we know, the glibc cannot be compiled by the LLVM/Clang compiler. So, we leverage the outputs of the GCC compiler, which can be generated during the compilation. A two-level indirect call analysis is employed to find out the possible targets of the indirect calls. It first leverages the type-based analysis to find out the possible address-taken callees. After that, a precise type-based callee analysis based on value flow tracking is employed to reduce the possible targets of indirect calls.

After constructing the control flow graph, the mapping between API arguments and syscall arguments is analyzed. Firstly, the data dependency graph is constructed by leveraging the backward taint analysis to find out the data sources of each syscall argument. If all of the data sources are determined (e.g., constants or API arguments), the possible value set of the argument can be analyzed. Our taint analysis starts from the syscall arguments and finds all of the data sources to construct the data dependency graph. During the analysis, all of the conditional judgment statements are extracted and added in the

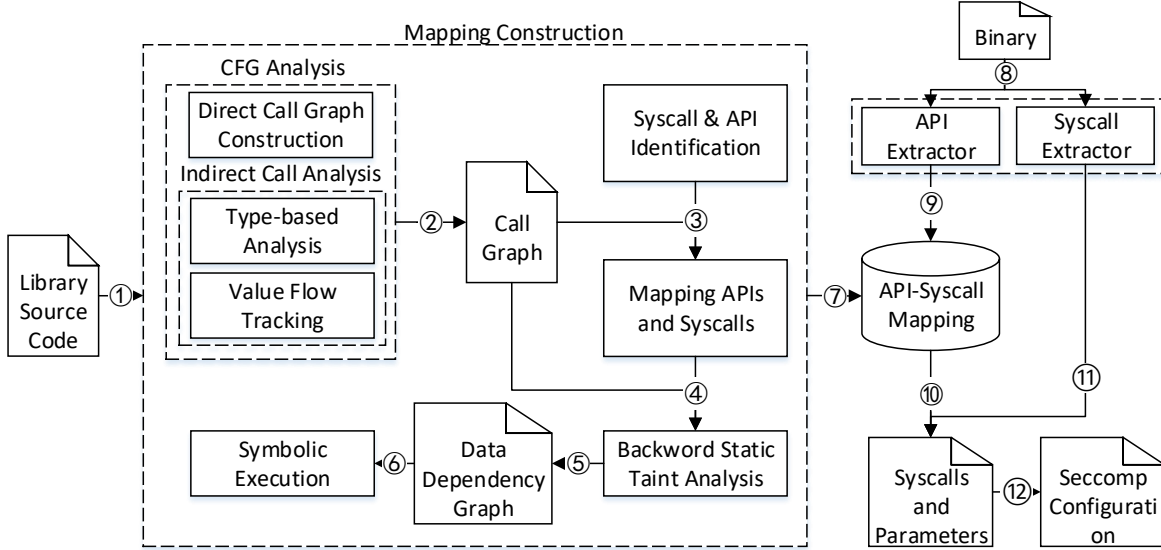


Fig. 2. The workflow of our system. ① The library source code is fed into the mapping construction module. ② The CFG Analysis module outputs the function call graph of the library. ③ By identifying the syscall and API points in the call graph, the mapping between APIs and syscalls can be constructed. ④ The backward taint analysis is used to find out the data sources of each syscall argument. ⑤ The data dependency graph is constructed during the taint analysis. ⑥ The symbolic execution is employed to analyze how the conditional judgment statements can affect the arguments in the data dependency graph. ⑦ Based on the results of taint analysis and symbolic execution, the mapping between API arguments and syscall arguments is constructed. ⑧ The target binary is analyzed to extract the dependent API and arguments. ⑨ The extracted API and arguments are used to search the mapping ⑩ for possible dependent syscalls and arguments. ⑪ If the binary does not use a dependent library to invoke syscalls, the syscalls and arguments are extracted. ⑫ Seccomp configuration with allowed syscalls and arguments can be generated.

graph. So that the variables that affect the value of the syscall argument can be obtained. To find out the final argument value set, the symbolic execution is employed to analyze how the conditional judgment statements can affect the arguments in the corresponding code blocks or functions. Finally, the mapping between API arguments and syscall arguments is constructed.

Next, the binary analysis is employed to determine the dependent library APIs and arguments of a target binary. The invocations of library APIs can be obtained by disassembling the target binary. We adopt the API argument analysis approach of [23] to obtain the arguments of the dependent APIs at best effort. Based on the mapping and dependent APIs, the dependent syscalls and arguments can be obtained. Finally, Seccomp configuration with allowed syscalls and arguments can be generated.

In Linux environments, most binaries invoke syscalls through dependent libraries, but some binaries invoke syscalls through the `syscall()` API, or embedded assembly code directly. When a binary invokes syscalls through the assembly instructions (i.e., the syscall instructions), the analysis approach of [23] can be employed to get the syscall number and the corresponding arguments, which analyzes the syscall arguments through binary analysis. For the cases of invoking syscalls through the `syscall()` API, the proposed method of [23] can be used to find out the syscall name and arguments. If the API invocation is performed through an indirect call, our system analyzes it in a best effort way, since it is an opening problem to analyze indirect calls in binaries. The target API destination is searched backward from the API callsite using the method of [23]. If the API target cannot be obtained, the

binary cannot be protected, and we leave this problem in the future work. Fortunately, this case is not common in our target binaries, and we did not encounter this case in the evaluation.

B. Mapping APIs with Syscalls

This stage aims to construct the mapping between library APIs and syscalls by constructing the control flow graphs of the dependent libraries. To that end, the direct calls are analyzed with the assistance of the compiler and the indirect calls are analyzed by a two-level analysis approach. Based on the full control flow graph, the mapping can be constructed.

1) *Direct Call Graph Construction*: The LLVM compiler is widely used to generate the function call graphs of open-source software at compiling time. However, LLVM does not support the glibc compilation, so we need to use some other methods. Intuitively, we can construct the function call relationships from the C source files. But, C source code can be very complicated and it is difficult to parse them. On one hand, macros are often used in C source files. When parsing, it is inevitable to replace macros. There are many kinds of macros and their structures are complex. Furthermore, they are often accompanied by deep nesting. On the other hand, there are many functions with alias names in the C source code, which can cause many difficulties for our analysis. Therefore, how to analyze the source code automatically is a challenge.

Another problem of the source code analysis is that the analysis of the syscall invocation is library-specific. Some syscalls are invoked through assembly code in the source file, so only analyzing the C code is not enough. As described in the glibc wiki [25], some syscalls are invoked through an assembly wrapper, and the compiler will directly convert a

function name into the assembly code that invokes the syscall. So, the analyzer cannot find the syscall instructions of these syscalls in the source code.

To address these challenges, we build a direct call graph based on the disassembly of the glibc library. In the disassembly file, we can obtain the instruction information contained in each function. When the callsite is executed, its operand may be an immediate value or a register. If the operand is an immediate value, the immediate value is the offset of the function in the binary file, and the function name of the called function is enclosed by “<>” at the end of the line. According to this rule, we can easily know which functions are directly called by a function. However, when the operand contains registers (register+immediate or only registers), it is an indirect call, which is analyzed later.

2) *Indirect Call Analysis*: The indirect call analysis is an opening problem in static analysis. In this paper, we adopt the two-layer indirect call analysis of [24], [26] to find the over-approximated callees of the indirect calls. The two-layer analysis first finds all of the address-taken functions. The address-taken functions are the functions which addresses are stored in global variables or objects. These addresses are usually used as the targets of indirect calls. Then, the type-based alias analysis [27]–[29] can be used to identify the possible callees, by comparing the return types and arguments types between the callsites and callees. This approach can only find possible callees in a coarse-grained manner. We leverage the fine-grained type-based analysis approach [26] to further reduce the possible callees. When an indirect callsite fetches a function address from an object, only the functions which addresses are stored in the same object type can be the possible callee. Based on this observation, we collect the detailed information of the objects that store the address-taken functions. The possible callees can be obtained by matching the corresponding object type.

3) *Generating the Mapping*: The comprehensive control flow graph can be constructed by combining the direct call graphs and the indirect call graphs.

After constructing the control flow graph, the functions that invoke syscalls should be identified. There are three ways for the glibc to invoke syscalls [25]. The first method is to wrap the API invocation to the syscall invocation, and the wrapper is in the syscall-template.S. After the compilation, the corresponding APIs will be implemented using the assembly code with different syscall numbers. The list of such APIs is kept in the source code of the glibc. The second way is to invoke syscalls through macro functions, such as “SYSCALL_CANCEL”, which are also defined in the source code. The third method is to leverage the assembly code. Some functions in the glibc are programmed with the assembly code instead of the C language. The assembly code usually invokes syscalls. To determine if the functions is syscall-related, the assembly analysis is employed.

Next, the API function are identified in the control flow graph, which names are collected from the glibc document [25]. By searching the control flow graph by starting with the API functions, the reachable syscalls of the API are collected. So that the API-Syscall mapping is constructed. However,

this mapping is coarse-grained, which does not contain the argument relationship between APIs and syscalls.

For the binaries that invoke syscalls through the swi instructions, a binary analysis approach is proposed. There are two ways to pass the syscall number. The first way is to pass the syscall number by the swi instruction itself, so the number can be obtained through analyzing the instruction. The other way is to pass the syscall number based on the r7 register, which can be analyzed by using the static data flow analysis. The collected syscalls will be added to Seccomp configuration.

C. Mapping Arguments Between APIs and Syscalls

To further limit the arguments of syscalls, the dependent API arguments are firstly identified based on the static analysis. Then, the mapping between API and syscall arguments is generated. The analysis includes two steps: the backward static taint analysis and the symbolic execution. The fine-grained Seccomp configuration with syscalls and arguments can be generated based on the mapping and the dependent API arguments of the target program.

1) *Extracting API arguments*: We adopt the backward data flow analysis of [23] to obtain the API arguments of applications statically, which workflow is shown in Figure 3. It first pinpoints all of the API callsites in the target binary and then leverages the backward data flow analysis to extract the corresponding arguments at best effort.

Our system is designed for the ARM Linux platform, so there are some parts specially designed for the ARM platform. First, the syscall mechanism of the ARM platform is different from that of x86_64. In the x86_64 environment, the relationship between the syscall number and the syscall name can be found in “syscall_64.tbl”. However, there is no such file for the ARM64 environment, so “unistd.h” is used to establish a corresponding relationship. Taking the “read” syscall as an example, the syscall number is 0 in the x86_64 environment, while the syscall number is 63 in the ARM64 environment. Second, the inline assembly code used to invoke syscalls in glibc under the ARM platform is different from that under the x86_64 platform, so the binary analysis should be designed for the ARM platform. For example, the inline assembly code to invoke syscalls in x86_64 is “syscall”, while the assembly code is “svc 0” in ARM64. In addition, some functions in glibc are implemented in assembly code, which causes the analysis to be different under different architectures. Furthermore, the instruction analysis and the propagation analysis of ARM applications are different from those of x86_64 applications.

As illustrated in Figure 3, the binary analysis first extracts all of the callsites to the dependent library APIs. Data flow analysis is an opening problem in static binary analysis [30], so the system works in a best-effort way.

The analysis targets at three types of variables, which are: 1) constant values, 2) stack values and 3) register values. The backward analysis starts from the API arguments and tracks the value propagation through registers and memory. When the analysis reaches the beginning of a function, the parents of it in the control flow graph are continued to be analyzed. After the analysis, the result of an argument is either known

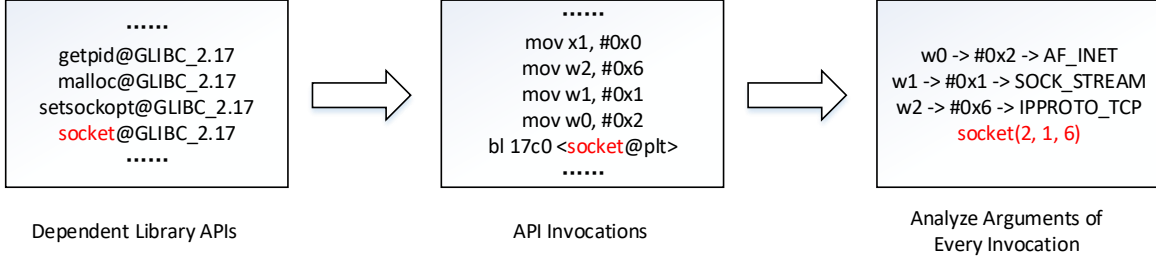


Fig. 3. An example to show the workflow of API argument extraction. In the example, the socket API was found to be used for syscall invocation. By analyzing the API invocation callsite, the w0-w2 were used for storing the API arguments. After analyzing the values of w0-w2, the API arguments (2,1,6) were determined.

and unknown. The unknown argument means the source of it cannot be determined.

There are several types of known arguments, which are: 1) flag arguments, 2) range arguments and 3) distinct value arguments. The flag argument is a combination of multiple flags using the logic OR operation. Each flag has its own meaning. For instance, the flag of the fopen API is combined with different flags (e.g., “r”, “w”, etc.). The range argument is a value with an upper and lower limit. For example, a memory address has its start address and the end address. The distinct value argument may take a specific value of a value set.

Fortunately, many API arguments are constants or fetched from a determined data set, which can be easily reached through backward data flow analysis. These arguments are usually the access modes, data sizes, protection flags, etc. The obtained API arguments are used to further configure Seccomp with the mapping of API arguments and syscall arguments.

2) *Backward Taint Analysis*: To determine the possible value of a syscall argument, a backward taint analysis approach is proposed to generate the data dependency graph of the argument. If all of the dependency variables are constants or API arguments, the data set of the syscall argument is determined. The taint analysis is employed because it can obtain the data dependency relationship without analyzing the complicated constraints in the control flow. In contrast, it is difficult to obtain the value set of a syscall argument from the API entries by symbolic execution. There are many variables that may affect the execution paths, and some of them cannot be analyzed statically. But, the taint analysis can skip these constraints. The taint analysis starts from the syscall arguments instead of API arguments, because we aim to construct the data dependency graph of every syscall argument. Some syscall arguments are assigned by some constants near the syscall invocation functions, which are difficult to analyze by tainting the API arguments.

The backward taint analysis takes the control flow graph of a library API as the input and outputs the data dependency graphs of every dependent syscall argument. Since Seccomp cannot filter pointers, so the analysis only analyzes the arguments with non-pointer types. For a syscall in the control flow graph of a library API, the non-pointer arguments are set as the taint source. For every argument, the variables that are

passed to the tainted variables are set as tainted. If the tainted variable is calculated from other variables, these variables are set as tainted, and the corresponding calculation statements are recorded. When the taint analysis reaches the function arguments, the parent functions in the control flow graph are continued to be analyzed, and the corresponding arguments in the parent functions are set as tainted. The taint sinks when the tainted variable is a constant, an API argument or an object field. By combining the backward data flow propagation process and the recorded statements, the data dependency graph is constructed.

If all of the data resources of a syscall argument are determined, the syscall argument is determined.

The backward taint analysis is inter-procedural, so it suffers from the path explosion problem due to the indirect call analysis. The indirect call analysis is employed during the control flow graph construction, which is an opening problem in static analysis. As a result, the control flow graph is over-approximated. Since our taint analysis follows the control flow graph, the analysis suffers from the path explosion problem. To mitigate this problem, two approaches are applied. First, we leverage state-of-art indirect call analysis approaches [24], [26] to reduce the false positives. Second, we have optimized the taint analysis process to make it faster by recording the results of the taint analysis. When the taint analysis analyzes a function for the second time, the tainted variables can be directly obtained according to the records. These approaches cannot solve the problem completely, and we leave this problem in future work. But, the impact of this issue is limited in this paper. The over-approximated taint analysis result will make API arguments correspond to more syscall arguments, so that some unused arguments will be allowed. In the worst cases, our approach cannot obtain the mapping between API arguments and syscall arguments. So, this problem can only affect the capability of argument filtering, and the over-approximated results will not affect the normal execution of the target program. But, our system can still resist the exploitation of many CVEs as described in the evaluation.

3) *Symbolic Execution*: To further determine the relationship between API arguments and syscall arguments and shrink the possible syscall arguments at best effort, the symbolic

execution is employed, which analyzes the functions that contain the possible data sources. The main idea is to obtain the relationship between the function arguments and the in-function constants that could be the data resources of a syscall argument. If the possible constants of a syscall argument are only determined by an API (or function) argument, the relationship can be obtained.

The symbolic execution depends on the intermediate files generated by the gcc compiler. Before starting to find the argument relationship, we first create a control flow graph of each related function. The control flow graph of a function takes a statement as a node, and a jump statement is a directed edge. A directed edge starting from a non-jump statement (such as an assignment statement) points to the instruction following it. A jump statements may emit one or more directed edges (such as goto and switch statements) to point to other statements. Starting from the first statement of the function, we can simulate the execution of the function.

Before the execution starts, we first assign a special symbol to each argument of the target function. The execution process starts from the first statement of the function. When processing an assignment statement, we record the value assigned to the variable. If the variable is used by the conditional statements (e.g., if or switch), the constraints are analyzed to get each branch's condition. If the data assignment of a syscall argument is only determined by the function's arguments, the mapping is constructed. After the execution, if all of the data sources of a syscall are determined by the function arguments, the data set of the syscall argument can be reduced. If a function argument is identified as the decisive factor of a syscall argument, the argument is set as tainted and the backward taint analysis is employed to further analyze the data sources of it. The most complicated case that our system has handled is converting a string to an integer flag. In this case, the flag is only determined by each character of the string, so the mapping between strings and flags can be constructed.

During the symbolic execution, we find that the same function may be analyzed for many times. Some complicated functions will make the analysis extremely slow. To improve the performance, we record the analysis result of every function, so that there is no need to analyze the function again. In addition, when we analyze a branch, we analyze the two branches at the same time to improve the performance. The analysis records the symbolic correspondence between the parameters of each function and its sub-function parameters in memory objects. The symbolic relationship are stored as strings. The static analysis is performed offline with a powerful server and generates configuration files for the embedded devices, so the analysis overhead is acceptable and will not affect the scalability.

The ability to simulate the execution of functions benefits from the simplicity of statements of the intermediate output by gcc. It is very complicated to simulate the execution of C language, but the statements in the intermediate file are simpler. For example, there are several situations in an assignment statement: conversion, calculation and function calls before the assignment. Fortunately, these problems can be addressed by analyzing the intermediate outputs of the

compiling, which can present the operations step by step.

The symbolic execution faces many problems, such as path explosion and difficulty to solve some complicated constraints, which make it difficult to apply it to complex programs. In our approach, symbolic execution is applied in dynamically-linked library functions that contain the possible data sources, and in most cases there is only numeric transfer. Although symbolic execution cannot handle particularly complex cases, our application scenario (i.e., argument transfer in library functions) are relatively simple. In this case, the flag is only determined by each character of the string, so the mapping between strings and flags can be constructed. In the worst cases, the mapping between API arguments and syscall arguments cannot be constructed, and this problem can only affect the capability of argument filtering. So, the impact of the problems of symbolic execution is acceptable.

4) *Seccomp Configuration Generation*: The mapping between API arguments and syscall arguments is added to the API-Syscall mapping constructed in Section III-B. So, the final API-Syscall mapping can be used to find not only an application's dependent syscall set but also the corresponding arguments in a best-effort way. Based on the dependent syscall list and some arguments, Seccomp configuration can be generated. Since the kernel attack surface is reduced by the embedded kernel security module, the introduced overhead will be lower than self-written tools [23].

IV. IMPLEMENTATION

In this section, we will discuss some implementation details. The prototype system proposed in this paper is implemented based on glibc v2.31 and implemented under the ARM structure.

A. Compilation

The main idea of the system is to analyze the relationship between functions, including the function call relationship and the argument transfer relationship between functions. However, the source code is not easy to analyze directly due to many factors, such as macros, etc. In order to facilitate the analysis, the gcc compiler is employed to generate semantic dumps during the compilation. When using gcc to compile the glibc source code, in addition to using the flags to add the debugging information and the optimization flags, two flags for generating special files are also used, which are “-fdump-ipa-cgraph” and “-fdump-tree-cfg”. After the compilation, we can obtain the “.cfg” file and the “.cgraph” file corresponding to each “.c” file. The cfg file uses a simple syntax to describe the functions defined in the c source file, and the cgraph file contains the alias information of the function. With these two files, the function relationship analysis becomes easier.

B. Indirect-call analysis

Inspired by [24], [26], the indirect calls are analyzed by the two-level analysis approach, which first identifies the indirect callsites. To this end, the compiling dumps of the dependent libraries are used. These dumps are generated during the

compiling, so they contain the semantic information, such as functions, code blocks and statements. If the operand of a callsite is a variable, the callsite is identified as an indirect call.

To identify the address-taken functions, we check if the operand of each statement is a function. Besides, if a function name is used as the argument of another function, the function is also identified. In this step, the mapping between function pointers and objects is constructed. If a function address is stored in an object, a two-tuple (function, object type) will be recorded. This is based on the observation that the targets of indirect calls are usually fetched from objects, and the addresses are stored in the objects previously. So, only pointers stored in objects of the same type can be the targets of indirect calls.

For each indirect call, the possible targets are first identified through the type-based alias analysis from the address-taken functions. The type-based alias analysis compares the number and types of the callsite's arguments with those of address-taken functions. To this end, we need to collect the argument types of every callsite and address-taken function. The function arguments are defined in the function definition. To identify the types of callsite arguments, the definitions of them are analyzed through the backward data analysis. Based on the observation that the arguments are declared in the same function of the callsite, the types of them can be easily collected. By comparing the argument types of callsite and callees, the possible targets can be identified. As discussed above, only pointers stored in objects of the same type can be the targets of indirect calls. So, if the function address is fetched from an object, the type of it is collected. The possible targets are limited to the functions with the same object type in the mapping.

C. Symbolic Execution

The symbolic execution is performed within functions, and it follows the function control flow. When a function is invoked, the argument information of the function is analyzed. If the argument is a variable name, we replace the variable name with the content of a variable (i.e., a symbol). If the argument is a number, we use the number as the symbol directly. For other types of values, we do not process them for now. For a function, if the value of a certain argument belongs to a finite set of numbers, the value range of the argument is determinable. After we analyze the functions following the control flow graph, we can further analyze the data relationship layer by layer through function calls.

The symbolic execution also runs at best effort, since blocking a syscall argument incorrectly may make the program crash. So, if we find that the value of a syscall argument cannot be determined, the argument will not be restricted.

V. EVALUATION

In order to evaluate our prototype system, we select 100 programs from the ARM Linux, which are used to test the effectiveness and performance of our system. To obtain the analysis targets, we have collected 1,208 ARM firmware from

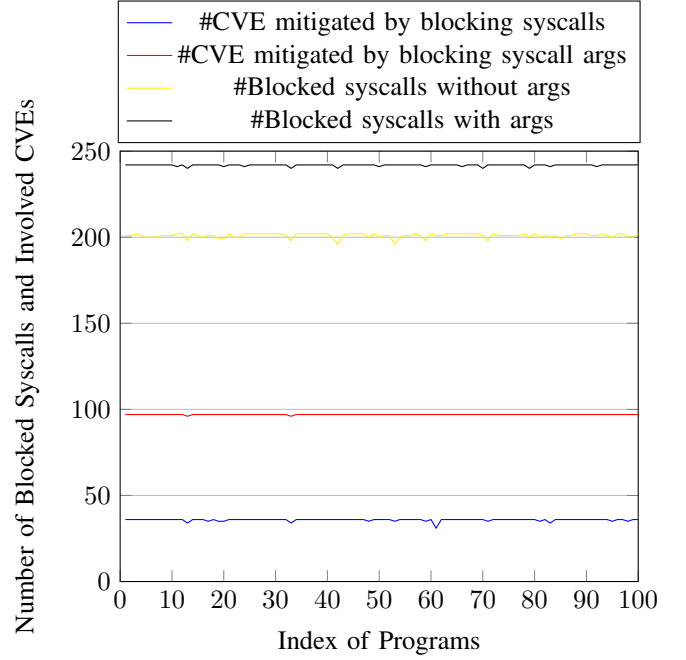


Fig. 4. The statistics of the syscall dependency analysis on every program, and the number of CVEs involved.

the dataset of Firmadyne [31], and leveraged binwalk [32] and Firmwalker [33] to unzip the file images. From the unzipped file images, we selected 100 common glibc-based programs for the evaluation, which are mainly in the /bin/, /user/bin and /sbin/ file directories. There are 2 criteria for our selection of target programs. First, the target program should be included in most (more than 90%) images. Second, the program should be written in C language and based on the glibc, since our approach leverages glibc and C programs as representative analysis targets. The average size of the selected programs is 71.7 kB.

A. Effectiveness

First of all, we analyze whether Seccomp restrictions imposed by our system could affect the normal execution of the target programs. Then, we test the effectiveness of blocking unused syscalls and syscall arguments.

The Seccomp profile generated by our system can block some syscalls and syscall arguments of a target program. However, if one of the dependent syscall or argument is restricted incorrectly, the target program may crash. Therefore, the set of non-disabled syscalls and arguments generated by our system must be over-approximated.

We analyze the target programs to generate Seccomp profiles, including syscall limitation and syscall argument limitation. The experimental results are shown in Figure 4. In the figure, the x-axis is the index of the program, and the y-axis is the number of syscalls or CVEs. In this part, we focus on the yellow and black lines. The yellow line indicates the number of syscalls disabled for the programs, and the black line indicates the number of affected syscalls with arguments. The part between the black line and the yellow line is the number of

syscalls whose arguments are restricted. The syscalls involved in the experiment are extracted from the `arch-syscall.h` header file, which lists a total of 291 syscalls. According to the experimental result, each program has 201 syscalls disabled and 40.82 syscalls with restricted arguments on average, which can significantly reduce the attack surface of kernel.

To verify the correctness of syscall limitation, we leverage the strace to track the execution of each program and record the invoked syscalls of them. All of the invoked syscalls and syscall arguments are within the analysis results, which proves that our system can correctly obtain the over-approximate set of the dependent syscalls and arguments.

B. CVE Mitigation

We evaluate the system's capability of CVE mitigation. If a syscall that contains a CVE is blocked by our system, the corresponding CVE is also mitigated. So, we collect the mapping between CVEs and syscalls from `cve.org` to test the capability of CVE mitigation. We check the descriptions of CVEs from 2020 to 2013, and filter out the CVEs whose descriptions clearly contain the keywords "syscall" or "system call". After that, we further select those CVEs that clearly indicate the affected syscalls and arguments, which number is 120. Based on the results, the mapping can be constructed.

Next, the CVEs that can be mitigated for each program can be analyzed. For each collected CVE, the corresponding syscalls and arguments can be obtained. Then, we check the allowlist of each program to find out if the CVE-related syscalls or arguments are blocked by the Seccomp configuration. If the CVE-related syscalls or arguments can be blocked, the syscalls, arguments and the corresponding CVE are recorded. The results are shown in Figure 4.

As shown in Figure 4, the red line is the number of CVE mitigation by blocking syscalls and syscall arguments, and the blue line is the number of CVE mitigation by only blocking syscalls. The part between the blue line and the red line is the number of newly mitigated CVEs by limiting the syscall arguments. On average, 35.79 CVEs are mitigated by blocking syscalls for one program, and 61.19 CVEs are newly mitigated by restricting syscall arguments.

From the results, we can find that blocking the accessible syscalls can effectively mitigate CVEs, and further limiting the syscall arguments can mitigate more CVEs. In addition, the number of new CVEs affected by limiting the arguments is larger than that of only blocking syscalls, which shows that further limiting syscall arguments is very meaningful.

Table II illustrates the top 20 syscalls with the related CVEs, which are blocked or restricted with arguments by our system for the target programs. From the table, we can find that the network-related syscalls introduce the most CVEs. There are 120 CVEs in the CVE collection. After we combine the CVE collections mitigated by different programs, 97 of them can be mitigated.

Taking the CVE-2017-7308 as a case study, where PoC is given in exploit-db [34]. The vulnerability locates in the `packet_set_ring` function in the `net/packet/af_packet.c` file. When the `packet_set_ring` function encounters the

`TPACKET_V3` flag, it will check the size of the private area (i.e., `tp_sizeof_priv` member variable) of the ring buffer block set in the user request. However, the subtraction of unsigned numbers is used in the check and the result is forcibly converted to the int type, which leads to some huge `tp_sizeof_priv` values that can bypass this check. By carefully setting the value of `tp_sizeof_priv`, the attackers can cause out-of-bounds writes to the kernel heap when receiving data packets. Besides, the offset of writing can be controlled, and the KASLR can be bypassed. After that, the SMAP and SMEP are disabled and the kernel heap is reshaped, so that the block of the ring buffer could overwrite the `packet_sock` structure and the function pointer field in it can be overwritten. Finally, the `commit_creds` function is executed in the context of the user process, and the root privileges can be obtained. The attack vector depends on two syscalls (i.e., `socket` and `setsockopt`) and the `AF_PACKET` flag. If a program does not rely on this flag in normal execution, the flag can be blocked by Seccomp, so that the vulnerability can be mitigated. Shredder [23] can analyze the library API arguments that may be used by an application. By using Shredder, the dependent APIs and corresponding arguments of a web-related application can be obtained. After that, the corresponding syscall arguments can be determined and blocked by our system. If an application (such as telnet) does not rely on the `socket` syscall with the `AF_PACKET` argument, the argument can be limited by Seccomp. Even if the application is hijacked by an attacker in some way, the attacker cannot exploit this CVE to crash the system or gain root privileges.

Through our observation, some syscalls are easy to restrict, but others are difficult. For instance, the network-related syscalls, such as `sockets`, have a very short call chain from the related API to this syscall. Many arguments of the API are directly passed to syscalls for use. For this kind of syscalls, even if they cannot be blocked, there is a high possibility to limit their arguments. And, the CVEs related to these syscalls are also easy to mitigate. However, there are many call chains from APIs to syscalls that are very long, which makes it difficult to determine the arguments of syscalls. In this case, if the program uses the API, the corresponding syscall arguments cannot be blocked, so the CVE related to the syscall cannot be mitigated. Some syscall arguments depend on global variables/pointers in the code implementation, which cannot be handled currently. In addition, if the syscall arguments dependent on complicated data structures, the data source cannot be analyzed by our system for now. And, we leave these problems in the future work.

C. Performance

Our system leverages Seccomp to secure system kernel, the main factor that affects the performance of the target programs is Seccomp. Seccomp is an embedded module in Linux operating system. Seccomp only works when a program configures Seccomp and invokes syscalls. If a program does not configure Seccomp policies, the performance of syscalls will not be affected. So, Seccomp can only introduce overhead to programs that are configured with Seccomp. In this

TABLE II
TOP 20 SYSCALLS AND THE RELATED CVEs MITIGATED BY SYSVERIFY.

Syscalls	Number of CVEs	Representative CVEs
recvmsg	29	CVE-2013-3228,CVE-2013-3225,CVE-2013-7267
recvfrom	29	CVE-2013-3224,CVE-2013-3223,CVE-2016-10229
socket	18	CVE-2016-10200,CVE-2017-7277,CVE-2013-7339
setsockopt	11	CVE-2017-6074,CVE-2017-6346,CVE-2017-16939
bind	9	CVE-2016-10200,CVE-2017-7277,CVE-2013-7339
sendmsg	5	CVE-2017-9242,CVE-2016-3841,CVE-2018-1130
mmap	5	CVE-2017-14497,CVE-2018-7740,CVE-2016-4794
ioctl	5	CVE-2020-10942,CVE-2013-2239,CVE-2017-18257
sendto	5	CVE-2017-15115,CVE-2017-7308,CVE-2015-2686
ptrace	5	CVE-2013-0871,CVE-2014-3534,CVE-2014-4699
write	3	CVE-2017-7277,CVE-2017-7495,CVE-2015-8019
accept	3	CVE-2015-8970,CVE-2017-8890,CVE-2017-9075
writev	3	CVE-2014-0069,CVE-2016-9755,CVE-2015-8785
getsockopt	3	CVE-2017-15115,CVE-2013-1828,CVE-2013-4588
connect	3	CVE-2017-7277,CVE-2016-9755,CVE-2017-8824
read	2	CVE-2017-7495,CVE-2013-6432
unshare	2	CVE-2017-15115,CVE-2014-7975
madvise	2	CVE-2015-7312,CVE-2014-8173
syncfs	2	CVE-2019-19448,CVE-2019-19813
brk	1	CVE-2020-9391

section, we test the performance impact of it. The target programs used in the test come from the lmbench performance test tools. We test the performance from three aspects: memory read and write, file read and TCP bandwidth.

We use the API provided by the libseccomp library to configure Seccomp for the target programs and adopt a whitelist strategy. The Seccomp rules are automatically generated by our system. For example, if our system finds that the target program may use the socket syscall, and the first argument can only be AF_INET, then the system will generate a corresponding Seccomp rule. This rule will enable the program to successfully pass the check when calling the socket syscall with the AF_INET flag as the first argument. The LMBench is selected as the benchmark to test the performance. For comparison, we also test the performance of the approaches that only limit syscalls (e.g., Confine [11]) with the same testing tools and methods.

The results of the benchmark with different security policies (e.g., without Seccomp, with Seccomp limiting syscalls, with Seccomp limiting syscalls and arguments) are shown in Table III. To analyze the performance impacts, we calculate the average performance degradation on various metrics by limiting only syscalls and limiting syscalls and arguments. According to the calculation results, in the case of only limiting syscalls, the average performance of bw_mem decreases by 0.23%, bw_file_rd decreases by 0.69% on average, and bw_tcp decreases by 5.28% on average. When limiting both syscalls and arguments, there are similar results, 0.22%, 0.68%, and 6.95%, respectively. From the results, we can find that our approach introduces less than 2% more performance degradation than other Seccomp-based approaches that only limit syscalls. In some cases, the performance with Seccomp limitation is higher than that without Seccomp limitation, which means the performance fluctuation is higher than the performance impact of Seccomp limitation.

The performance can be further optimized by changing the order of Seccomp rules. After enabling Seccomp, an invoked syscall is matched with each Seccomp rule in turn until the

match is successful or all rules are not matched. So, the order of the Seccomp rules is important for the performance. If the order of Seccomp rules is not appropriate, it is possible that every syscall invoked by the target program will be matched for a long time. So, the order of the rules can be adjusted to further reduce the performance loss. A straightforward way is to use the strace to track the execution of the program, and rank the syscalls that are used frequently. Based on this order, the performance of Seccomp can be optimized.

VI. RELATED WORK

A. Lightweight VM-based Containers

It is a major way to leverage the lightweight virtual machines (VMs) to isolate applications, because VMs are considered to be more secure than containers and sandboxes [35]. Many lightweight VMs are proposed to protect the operating systems from applications. The unikernel [36] compiles the applications with the library OS [13] into a lightweight VM image, so that the image can be executed as a lightweight VM. Since the OS is integrated into the applications, the OS is not isolated from the user space, which is not secure. In addition, all of the applications should be recompiled or redesigned. The Kata Container [37] leverages VMs to isolate containers by running a container in a lightweight VM, which starts fast and consumes less resources. The goal of the Kata Container is to improve the isolation of the containers to the level of the virtual machines while maintaining the performance of the containers. However, the performance of it is lower than that of a Docker container. The gVisor [38] leverages the para-virtualization to isolate containers, which has two parts: Sentry and Gofer. Sentry emulates a virtual kernel for containers, which can handle most of syscalls invoked by containers. Therefore, the attack surface of the kernel is reduced. Gofer can redirect the I/O requests of containers to the host operating system. Compared with the Kata Container, the isolation of Gvisor is weaker, since some syscalls are handled by the host operating system.

TABLE III
PERFORMANCE COMPARISON IN THREE CASES: NO LIMITATION, LIMITING SYSCALLS, LIMITING SYSCALLS & PARAMETERS.

Benchmark	Block/packet size	Benchmark Mode	No limitation	Limiting syscalls	Limiting syscalls & params
bw_mem	32k	bcopy	44265.02	44257.37	44240.37
		bzero	88737.19	88674.75	88693.81
		fcop	11167.81	11175.11	11173.21
		frd	13375.55	13365.82	13371.43
		fwr	22309.37	22306.72	22289.05
		rdwr	44567.34	44559.24	44567.34
	64k	bcopy	43674.58	43694.70	43658.22
		bzero	89058.38	88953.72	89080.88
		fcop	11125.16	11124.86	11122.48
		frd	13361.36	13366.21	13375.91
		fwr	22310.13	22322.05	22328.74
		rdwr	43960.26	43971.15	43960.26
	128k	bcopy	39128.11	39302.49	39326.51
		bzero	86338.05	88660.29	88471.23
		fcop	11033.64	11027.64	11024.30
		frd	13214.15	13213.55	13221.13
		fwr	22336.87	22309.62	22316.70
		rdwr	30929.77	31001.92	30981.99
bw_file_rd	128k	io_only	9235.10	9104.86	9149.07
		open2close	8465.41	8375.36	8384.34
	256k	io_only	9216.76	9173.34	9165.17
		open2close	8851.62	8742.99	8741.35
	512k	io_only	8639.44	8567.75	8687.13
		open2close	8352.87	8325.10	8304.56
	1m	io_only	8140.52	8106.05	8122.64
		open2close	7983.59	7929.11	7964.36
	2m	io_only	7867.13	7850.30	7751.44
		open2close	7666.43	7649.65	7660.83
bw_tcp	128	\\	343.79	350.01	357.38
	256	\\	666.54	662.68	678.05
	512	\\	1139.12	1230.59	1250.85
	1024	\\	1953.47	2075.29	2106.87
	1437	\\	2343.87	2572.09	2610.74

In summary, a big problem of applying VM-based isolation approaches in IoT scenario is that most of the IoT/embedded devices do not have enough resources to support VMs, so some lightweight security enhancement approaches are needed.

B. Securing Operating Systems

The operating system is not secure in multi-application systems. Bastion [39] proves the weaknesses of the Linux network isolation, and performs cross-process attacks on the Linux system. [40], [41] show the weakness of the isolation of the proc file system. By using it, a process can obtain the information of other processes on the same host.

To secure the operating systems for multi-application systems, reducing the attack surface of the operating system is an important way. [11], [14]–[16] filter the unused syscalls for user-space applications, so that it is difficult for user-space applications to exploit the kernel vulnerabilities. Based on the observation that an application usually invokes different syscalls in different execution stages (e.g., initialization stage, servicing stage and exiting stage), [42] blocks different syscalls according to the application's execution stages. To this end, it identifies the dependent syscalls of different running stages. But, these approaches are coarse-grained, which can only limit the syscall numbers. In addition, these systems are designed for cloud services, and some of them introduce additional security monitors in the target systems. Unlike these systems, SCONE [43] compiles the dependent system service code into the applications by leveraging the libOS, so that it

does not invoke syscalls. However, the recompilation makes the approach's versatility worse.

C. Program Debloating

Besides blocking the syscalls, removing the unused code from the applications and the dependent libraries is another way to secure the operating systems. Piece-wise debloating [44] only loads the necessary dependent libraries and replaces the unnecessary parts with null code. Nibbler [45] removes the unused code by analyzing the function call graph of the target application. LibFilter [46] removes the unused functions in the dependent libraries. In contrast, Razor [47] constructs the function call graph through dynamic tracking. However, these the program debloating approaches need to redesign the applications or dynamically-linked libraries, so it is not easy to apply the approaches to commercial systems widely.

D. IoT Security

IoT security [48]–[52] is a hot topic, which is related with device security [53], [54], network security [55], cloud security [56], [57], etc. This paper focuses on the system security of IoT devices, so we mainly discuss the IoT device security.

IoT device firmware is usually developed by low-level languages (such as assembly and C language), so coding or design bugs are inevitably introduced in the development process. Due to limited hardware resources, IoT devices usually lack necessary dynamic system defense measures such as CFI

(Control Flow Integrity), etc., which make attackers exploit vulnerabilities more easily. A number of studies have pointed out that the memory vulnerabilities caused by code injection attacks are common in firmware [53], [54], and hijacking the control flow of IoT applications through modifying the function return address is still a main threat [58], [59].

The limited software and hardware resources of IoT devices are one of the main reasons why IoT devices are more vulnerable. Therefore, many researches focus on securing IoT systems.

The main approach is to protect the integrity of the program control flow, which can ensure the integrity of the function return address to deal with control flow hijacking attacks. μ RAI [58] stores the effective return address collection of the functions in the non-writable memory area, and makes the function return addresses fetched from the collection to ensure that the return address will not be tampered with. Silhouette [59] is based on the “Shadow Stack” technology to defend against control flow hijacking attacks. It configures a memory protection unit to implement memory access rules to ensure that the program must return to a legal target address after the return instruction is executed. However, these approaches introduce runtime overhead to the target applications. In addition, the OS is not protected when the applications are compromised.

Many firmwares do not have operating systems. To overcome the problems that there are no system protections, some studies divide the firmware into different components to implement least-privileged isolation. EPOXY [53] identifies instructions that require high privileges through static analysis, and then provides stack protection, code and data area isolation. But it cannot perform process-level code isolation. MINION [60] leverages the MPU to replace the memory area accessible by the process during context switching, thus realizing the isolation of the process memory space. But, the method to divide the data and code areas cannot be applied to complicated applications. ACES [61] overcomes the shortcomings of the above two schemes. By automatically identifying and separating the minimum execution unit of the firmware, a more fine-grained authority can be identified.

VII. CONCLUSION

This paper proposes a novel static dependent syscall analysis approach for IoT applications, which can obtain the dependent syscalls and the corresponding arguments. So that, a fine-grained kernel access limitation can be performed for the IoT applications. To this end, the mapping between dynamic library APIs and syscalls is built through static analysis. A novel argument mapping construction approach based on backward taint analysis and symbolic execution is proposed to further map the arguments between APIs and syscalls. After obtaining the dependent library APIs and the corresponding arguments of an application, the fine-grained kernel access control policy can be generated. The experimental results show that the system can block the unused syscalls and arguments from the target programs and mitigate the CVEs included in the corresponding syscalls with acceptable overhead. In future

work, we are going to apply and test our system in other computing systems, such as containers.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China (No.2021YFB2012402) and the National Natural Science Foundation of China under Grants NO. 61872111.

REFERENCES

- [1] B. Liao, Y. Ali, S. Nazir, L. He, and H. U. Khan, “Security analysis of iot devices by using mobile computing: a systematic literature review,” *IEEE Access*, vol. 8, pp. 120331–120350, 2020.
- [2] C.-T. Li, C.-C. Lee, C.-Y. Weng, and C.-M. Chen, “Towards secure authenticating of cache in the reader for rfid-based iot systems,” *Peer-to-Peer Networking and Applications*, vol. 11, no. 1, pp. 198–208, 2018.
- [3] I. Yaqoob, I. A. T. Hashem, A. Ahmed, S. A. Kazmi, and C. S. Hong, “Internet of things forensics: Recent advances, taxonomy, requirements, and open challenges,” *Future Generation Computer Systems*, vol. 92, pp. 265–275, 2019.
- [4] Y. B. Zikria, S. W. Kim, O. Hahm, M. K. Afzal, and M. Y. Aalsalem, “Internet of things (iot) operating systems management: Opportunities, challenges, and solution,” 2019.
- [5] “Cve-2017-7308,” <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7308>.
- [6] “Cve-2017-5123,” <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5123>.
- [7] “Cve-2016-8655,” <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8655>.
- [8] “2020 iot developer survey,” <https://iot.eclipse.org/community/resources/iot-surveys/assets/iot-developer-survey-2020.pdf>.
- [9] M. Xu, C. Song, Y. Ji, M.-W. Shih, K. Lu, C. Zheng, R. Duan, Y. Jang, B. Lee, C. Qian *et al.*, “Toward engineering a secure android ecosystem: A survey of existing techniques,” *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, pp. 1–47, 2016.
- [10] “Linux seccomp,” https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.
- [11] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, “Confine: Automated system call policy generation for container attack surface reduction,” in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 443–458.
- [12] “Ibm. nabra containers: a new approach to container isolation.” <https://nabra-containers.github.io/>.
- [13] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, “Cooperation and security isolation of library oses for multi-process applications,” in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–14.
- [14] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, “Speaker: Split-phase execution of application containers,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 230–251.
- [15] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, “Mining sandboxes for linux containers,” in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 92–102.
- [16] S. Barlev, Z. Basil, S. Kohanim, R. Peleg, S. Regev, and A. Shulman-Peleg, “Secure yet usable: Protecting servers and linux containers,” *IBM Journal of Research and Development*, vol. 60, no. 4, pp. 12–1, 2016.
- [17] Y. Niu, L. Lei, Y. Wang, J. Chang, S. Jia, and C. Kou, “Sasak: Shrinking the attack surface for android kernel with stricter “seccomp” restrictions,” in *2020 16th International Conference on Mobility, Sensing and Networking (MSN)*. IEEE, 2020, pp. 387–394.
- [18] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, “Sysfilter: Automated system call filtering for commodity software,” in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 459–474.
- [19] Q. Zeng, Z. Xin, D. Wu, P. Liu, and B. Mao, “Tailored application-specific system call tables,” Tech rep., Technical report, Pennsylvania State University, Tech. Rep., 2014.
- [20] C. Canella, M. Werner, D. Gruss, and M. Schwarz, “Automating seccomp filter generation for linux applications,” in *Proceedings of the 2021 on Cloud Computing Security Workshop*, 2021, pp. 139–151.
- [21] C. Mulliner and M. Neugschwandtner, “Breaking payloads with runtime code stripping and image freezing,” *Black Hat USA*, 2015.

- [22] “The most important linux files to protect,” <https://www.beyondtrust.com/blog/entry/important-linux-files-protect>.
- [23] S. Mishra and M. Polychronakis, “Shredder: Breaking exploits through api specialization,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 1–16.
- [24] K. Lu, A. Pakki, and Q. Wu, “Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences,” in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1769–1786.
- [25] “System call wrappers - glibc wiki,” <https://sourceware.org/glibc/wiki/SyscallWrappers>.
- [26] K. Lu and H. Hu, “Where does it go? refining indirect-call targets with multi-layer type analysis,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1867–1881.
- [27] B. Niu and G. Tan, “Modular control-flow integrity,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 577–587.
- [28] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, “Enforcing forward-edge control-flow integrity in gcc & llvm,” in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 941–955.
- [29] R. M. Farkhani, S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi, “On the effectiveness of type-based control flow integrity,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 28–39.
- [30] X. Meng and B. P. Miller, “Binary code is not easy,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 24–35.
- [31] D. D. Chen, M. Woo, D. Brumley, and M. Egele, “Towards automated dynamic analysis for linux-based embedded firmware,” in *NDSS*, vol. 1, 2016, pp. 1–1.
- [32] “Firmware analysis tool,” <https://github.com/ReFirmLabs/binwalk>.
- [33] “Script for searching the extracted firmware file system for goodies!” <https://github.com/craigz28/firmwalker>.
- [34] “The exploit database - exploits, shellcode, 0days, remote exploits, local exploits, web apps, vulnerability reports, security articles, tutorials and more,” <https://www.exploit-db.com/>.
- [35] T. Combe, A. Martin, and R. Di Pietro, “To docker or not to docker: A security perspective,” *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, 2016.
- [36] A. Madhavapeddy and D. J. Scott, “Unikernels: the rise of the virtual library operating system,” *Communications of the ACM*, vol. 57, no. 1, pp. 61–69, 2014.
- [37] “Kata containers,” <https://katacontainers.io/>.
- [38] “gvisor: A container sandbox runtime focused on security, efficiency, and ease of use,” <https://gvisor.dev/>.
- [39] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin, “Bastion: A security enforcement network stack for container networks,” in *2020 USENIX Annual Technical Conference (USENIXATC 20)*, 2020, pp. 81–95.
- [40] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, “Containerleaks: Emerging security threats of information leakages in container clouds,” in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 237–248.
- [41] X. Gao, B. Steenkamer, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, “A study on the security implications of information leakages in container clouds,” *IEEE Transactions on Dependable and Secure Computing*, 2018.
- [42] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, “Temporal system call specialization for attack surface reduction,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1749–1766.
- [43] S. Arnaudov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumar, D. O’keeffe, M. L. Stillwell *et al.*, “Scone: Secure linux containers with intel sgx,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 689–703.
- [44] A. Quach, A. Prakash, and L. Yan, “Debloating software through piecewise compilation and loading,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 869–886.
- [45] I. Agadacos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, “Nibbler: debloating binary shared libraries,” in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 70–83.
- [46] B. Shtinfeld, “Libfilter: Debloating dynamically-linked libraries through binary recompilation.”
- [47] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, “Razor: A framework for post-deployment software debloating,” in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1733–1750.
- [48] M. A. Khan and K. Salah, “Iot security: Review, blockchain solutions, and open challenges,” *Future generation computer systems*, vol. 82, pp. 395–411, 2018.
- [49] V. Hassija, V. Chamola, V. Saxena, D. Jain, P. Goyal, and B. Sikdar, “A survey on iot security: application areas, security threats, and solution architectures,” *IEEE Access*, vol. 7, pp. 82 721–82 743, 2019.
- [50] W. H. Hassan *et al.*, “Current research on internet of things (iot) security: A survey,” *Computer networks*, vol. 148, pp. 283–294, 2019.
- [51] L. Xiao, X. Wan, X. Lu, Y. Zhang, and D. Wu, “Iot security techniques based on machine learning: How do iot devices use ai to enhance security?” *IEEE Signal Processing Magazine*, vol. 35, no. 5, pp. 41–49, 2018.
- [52] M. A. Al-Garadi, A. Mohamed, A. K. Al-Ali, X. Du, I. Ali, and M. Guizani, “A survey of machine and deep learning methods for internet of things (iot) security,” *IEEE Communications Surveys & Tutorials*, vol. 22, no. 3, pp. 1646–1685, 2020.
- [53] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, “Protecting bare-metal embedded systems with privilege overlays,” in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 289–303.
- [54] J. Powny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, “Cross-architecture bug search in binary executables,” in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 709–724.
- [55] B. Huang, A. A. Cardenas, and R. Baldick, “Not everything is dark and gloomy: Power grid protections against iot demand attacks,” in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1115–1132.
- [56] D. Zhan, L. Ye, H. Zhang, B. Fang, H. Li, Y. Liu, X. Du, and M. Guizani, “A high-performance virtual machine filesystem monitor in cloud-assisted cognitive iot,” *Future Generation Computer Systems*, vol. 88, pp. 209–219, 2018.
- [57] W. Zhou, Y. Jia, Y. Yao, L. Zhu, L. Guan, Y. Mao, P. Liu, and Y. Zhang, “Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms,” in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1133–1150.
- [58] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, “μrai: Securing embedded systems with return address integrity,” in *Network and Distributed Systems Security (NDSS) Symposium*, 2020.
- [59] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, “Silhouette: Efficient protected shadow stacks for embedded systems,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1219–1236.
- [60] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, “Securing real-time microcontroller systems through customized memory view switching,” in *NDSS*, 2018.
- [61] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, “Aces: Automatic compartments for embedded systems,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 65–82.