Towards an Automated Generation of Application Confinement Policies with Binary Analysis

Omitted for blind review

ABSTRACT

Application-based access control technologies are used to protect systems from malicious or compromised software. Existing rule-based access control systems rely on a comprehensive policy, which defines the resources an application is allowed to access. The generation of these policies is a hard and error-prone task for system engineers. In this work, we provide a framework to automate this task and a proof-of-concept implementation that uses binary analysis to generate a model of the resource requirements of an application. We use a new approach to refine the policy by connecting different accesses to the same resource via their least common ancestor in the call graph. Moreover, we tested the proposed methods with a common used webserver and show a high potential to significantly simplify the policy generation process.

Keywords

Security, Sandbox, AppArmor, Policy Generation, Privilege Classification

1. INTRODUCTION

Traditionally, access control systems have been based on the logged-on user. The system defines the privileges of a process based on the executing user's identity or roles assigned to this user. This approach relies on the claim that all applications act in the user's best interest. Due to programming errors, applications may be exploitable and do not meet this claim. Moreover, some applications are malicious by design (malware). Therefore, application specific confinement technologies, called sandboxes, have been introduced to enforce the principle of least privilege [1]. This means that an application is only allowed to access the minimum set of resources that are needed to execute correctly. Besides isolation-based systems, rule-based systems like *AppArmor* or *SeLinux* are available for major operating systems.

While these approaches have the potential to counter possible problems of malicious or compromised software, they rely on a comprehensive policy describing the privileges that should be granted to a software module. Determining these privileges can be a hard task. Software developers would usually have the expert knowledge which is necessary to build the confinement policy. However, they do not benefit from it and do not necessarily know the targeted system and intended use. The end user is the person who actually profits from a secure system, but might not have enough expert knowledge to understand the requirements of an application to the underlying system. In many cases there is a third party, the system developer, who combines different modules to one system. Automation and abstraction of policy generation would support all of these stakeholders.

Some approaches to scan applications for resource accesses and to generate confinement policies have been presented in the literature. However, most of them rely on dynamic tracing of the application behaviour and excessive test runs are needed to ensure that all code paths are executed [2, 3, 4]. Other approaches heavily rely on features of the used programming language or access to the source code [5, 6].

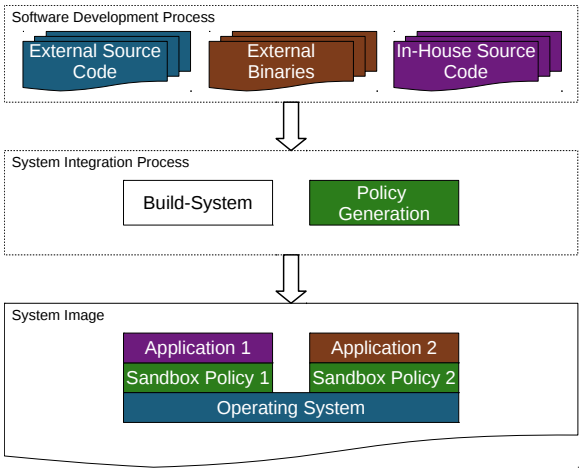


Figure 1: Illustration of an exemplary system development process: Software modules from different sources are integrated to the system image. The policy generation takes place in the integration step.

In this work, we aim to provide a generic architecture to ease the process of policy generation and create a high level description of the required privileges of software from different

sources. Figure 1 shows an excerpt of an exemplary system development process. Software modules from different sources are integrated to a system image, which provides an operating system with sandboxing capabilities and a number of user-space applications. In this approach, the previously defined system developer is in charge of collecting and generating the policies. This person combines knowledge of the used software components and the targeted application. The system developer may or may not have access to the full source code but has access to all binaries and a somewhat comprehensive documentation.

To identify the resource requirements of an application, we introduce a new approach that distinguishes between resource allocations and accesses on previously allocated resources. Connecting accesses and allocations provides additional information to refine the policy. Though this connection is currently done in a very simple way by finding the nearest allocation node of an access node in the program's call graph, analysis of widely used open source software shows good results in many cases.

We also provide a proof-of-concept implementation of the proposed system. The prototype generates and analyses a call graph based on an application's binary image. In combination with additional information like constant function parameters and user input it is able to generate confinement policies. We analyzed a common used web-server and compared the results to a runtime-trace and a working confinement policy. While we detected a good part of the privilege requirements, some problems remain for future work.

The paper is organized as follows. Section 2 discusses related work and Section 3 describes the proposed system. The implemented system is outlined in Section 4 and the results are shown in Section 5. In Section 6, the benefits and the drawbacks of the system, as well as future directions are summed up.

2. RELATED WORK

Application-oriented access control methods are widely used and part of many modern operating systems. Some methods to simplify and automate the policy generation have been presented. In the field of Intrusion Detection System (IDS), related approaches are used to detect run time anomalies by comparing the statically generated Control Flow Graph (CFG) or Call Graph (CG) with the run-time trace.

2.1 Application-Oriented Access Control

Basically, there are two different types of application-oriented access control [7]:

Isolation-based access control methods provide each confined application its own set of resources [8, 9]. Similar results can be achieved by using virtualization [10]. These schemes have some limitations which make it hard to use them for general purpose applications. Based on the design, each sandbox often has to have its own copy of resources and shared libraries [7].

In contrast to isolation-based systems, rule-based access control methods do not rely on an own copy of resources, but confine the access directly based on a policy. Common implementations are *Systrace* [11], *SeLinux* [12] or *AppArmor* [13]. These systems avoid many problems of isolation-based methods [7]. However, generating policies for different applications is still a difficult task. Coarsely grained policies (such as '*Allow access to file system*' or '*Allow access to network*') may allow too much and do not prevent possible threats. On the other side, fine grained policies (such as '*Allow read access to file X*' or '*Allow outgoing TCP connections to one server*') may result in complex policy management.

Usability and policy complexity is one of the biggest challenges for these systems. As shown in [14], usability has a high impact on the effectiveness of sandboxes. Automated generation of policies and policy abstractions help users to confine an application properly. Such abstractions have been introduced for *Mapbox* [15] and Functionality Based Application Confinement (FBAC) [16] and significantly improve the readability of policies by hiding unnecessary complexity.

2.2 Automation of Privilege Classification

Approaches to automatically generate parts of policies are widely spread. One method to suggest parts of the policy by analyzing linked libraries and desktop-entries has been presented for FBAC [6]. This method uses contextual information of libraries (an application which links against *libogg* might be a music player) and information in *.desktop* files in Linux systems, which often contain application categories (like *WebBrowser* or *Office*).

Systems like *AppArmor* or FBAC provide the functionality to trace application behaviour in test runs and use this information to generate or extend the policy. This method has the potential to provide a comprehensive policy, but it has to be ensured that all relevant execution paths have been triggered. Moreover, the collection of the policy information is usually done in an unconfined environment what might not be suitable, since the policy generation itself may be a threat to the test system.

In the field of IDS, system call tracing is used to detect abnormal application behaviour. *Paid* [17], for example, statically generates a call graph and uses graph inlining, system call inlining and source code injection to generate system call traces. *Paid* compares these traces with runtime traces to detect abnormal behaviour. In contrast to the approach presented in this paper, systems like *Paid* do not take parameters to resource accesses into account. Thus, they can not be used to prohibit an application for from accessing specific files.

Run-Time environments, such as Java and Common Language Runtime (CLR) provide stack based access control where stack inspection ensures that all calling methods are authorized to make privileged calls. For these systems, static approaches have been presented to automate policy generation [18, 19]. One approach [5] use dynamic analysis to refine statically generated policy models. The system is able to automatically generate test cases and asks the user to refine the statically generated policy stub. All these approaches use data flow analysis to classify resource requirements for applications, but rely on support of the underlying program language.

Recent work focusing on partition of software into least privilege components using technologies to detect the privilege requirements of single modules of the whole system. *ProgramCutter* [2] labels functions according to their system call invocations and uses data dependency to model the weight of the connection between two functions. It splits up the application according to their labels and enables big parts of the original code to run in a very restricted environment. *Passe* [3] is an extension of the Django web framework and uses data-flow and control-flow relationships to separate the privilege for so called *views*. A *view* is an automatically isolated component which represents the software module which is needed to serve one request. Both methods use dynamic learning technologies. Similar to learning mechanisms described above, they need excessive test runs which need to be executed unconfined.

3. SYSTEM ARCHITECTURE

3.1 Overview

We propose a framework to enable the mining of privilege requirements of applications from binary analysis and the generation of application confinement policies based on this information. The described framework and its current implementation is called *PolGen*.

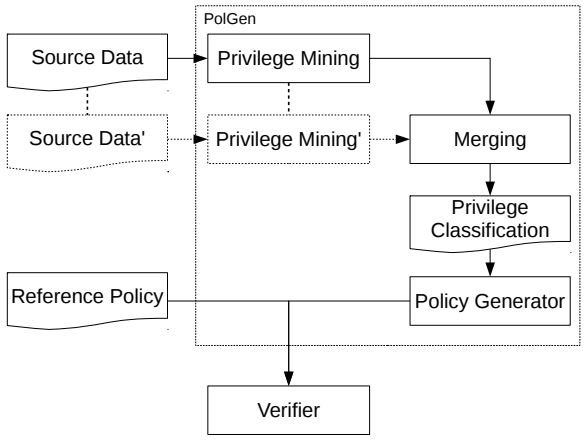


Figure 2: *PolGen*: Based on different types of source data, the privilege mining module generates a partial set of privilege requirements. These partial sets are merged to the privilege classification which is exported to an application confinement policy. Currently, this policy is verified by comparing it to a working reference policy of the given program.

The overall system architecture is shown in Figure 2:

- The *Source Data* currently represents the application binary and all linked libraries. However, *PolGen* is designed to use different sources like application source code or functional description to generate a privilege classification of the inspected application.
- *PolGen*'s first stage is *Privilege Mining*. Based on the type of the *Source Data*, the corresponding module is loaded and generates the *Privilege Classification* or parts of it.
- The *Privilege Classification* is an abstraction for application privileges. This abstraction enables the use of different mining and export modules.
- The *Policy Generator* generates a policy for the targeted sandboxing technology.
- The *Verifier* compares the generated policy to a working reference policy which is created manually or taken from current software distributions to verify *PolGen*'s functionality.

3.2 Privilege Classification

The privilege classification is a high level description of the resource requirements of the application. *PolGen* defines it as a set of *Privilege Requirements*. Each *Privilege Requirement* consists of three elements. A resource type, access types and additional parameters. The resource type defines the type of the accessed resource (e.g. *file* or *network*). Many resources may have different access types. Files, for example, may be accessed in read or write mode. Therefore, an access type is needed. Moreover, sometimes allowing access to all resources of a type is not desired. Therefore, additional resource properties are needed to minimize the granted privileges. These properties are represented as simple name-value pairs where the property list depends on the resource type.

3.3 Privilege Mining

The current version of *PolGen* provides a module to generate the privilege classification based on an application binary. The basic process, shown in Figure 3, is configured by the *Symbol Configuration* that provides the following information:

- Symbol[1] names for resource allocations and their corresponding resource type,
- symbol names for resource accesses and their corresponding resource type,
- symbol names which may have static parameters that give additional information of the resource access type, and
- blacklists for non-interesting symbols and syscalls

The first step is the generation of the call graph. Starting with *main*, each function f is represented as a node. Each possible transition (i.e., function call) T from f_1 to f_2 is represented as a directed edge. Self loops are ignored and multiple calls with the same source and destination node are merged because they do not provide additional information for the further analysis. Some function calls may have interesting static parameters which can be used later in the analysis. In this case, the edge gets the parameter as property. A good example is the *open* function in *libc*, which expects the access mode as second parameter. This access mode is often hard-coded. Thus, the *mov* instruction that sets the parameter contains a constant. The parameter can

[1]In this context, a symbol name corresponds to a function or system call name.

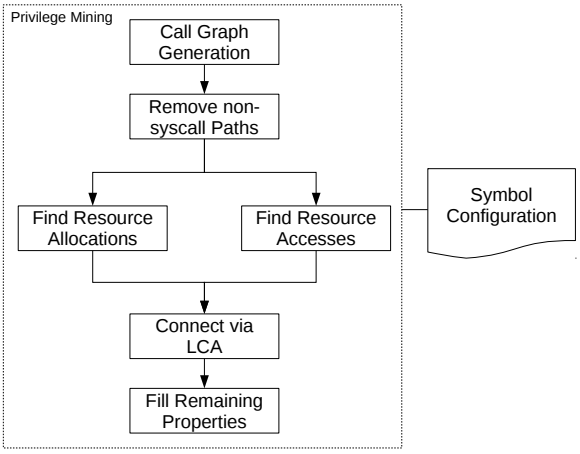


Figure 3: Privilege mining in *PolGen*: Function calls that do not lead to a system call are removed from the generated call graph. The resource allocations and accesses are located and connected via their LCA.

simply be parsed from the last *mov* to the parameter's address based on the compiler's calling convention. On *x86*, the value is stored in $0x4(\%esp)$, as shown in Figure 4. For performance reasons, *PolGen* performs this step on some pre-configured calls only.

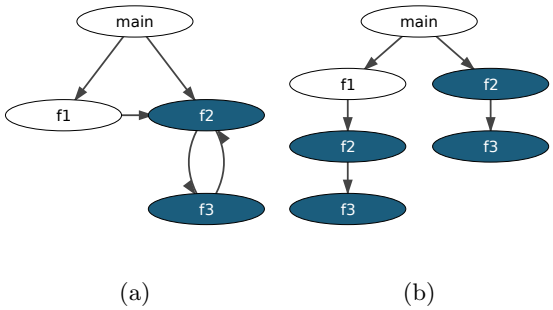
Similar to the parameter parsing, we identify system calls by searching userspace-kernel transitions and parsing the system call numbers based on the targeted architecture.

```
movl   $0x1a4,0x8(%esp) <open mode 0644>
movl   $0x1,0x4(%esp)   <O_WRONLY>
mov    0x8(%ebp),(%esp) <filename>
call   8048520          <call open>
```

Figure 4: Static function parameter detection for *gcc-x86*: *PolGen* is able to add the 'write' access mode to the given open-call since the *write only* flag is hard-coded.

The resulting graph contains one node for each called function, as well as one node for each system call in the application binary and all linked libraries. Each function call is represented as a directed edge. It is assumed that resource access is only possible via the kernel. Thus, all nodes which do not have a path to a system call are deleted. Additionally, some symbols and system calls may not be interesting at all. Thus, *PolGen* is instrumented with a blacklist which contains system calls like *getpid* to simplify the call tree and improve performance. For this graph simplification, we use a post-order tree visitor that deletes all non-syscall nodes which are blacklisted or do not have children.

To simplify the next steps, the directed graph is converted to a directed tree. As shown in Figure 5, we duplicate subtrees with multiple parents and eliminate cyclic subtrees. After the expansion, a node does not represent a function, but one call of this function. Consequently, the path from a node to the main node represents one possible stack trace for each function. This is necessary to correctly distinguish different resource allocations which are done with the same function.



(a) (b)

Figure 5: The call graph before (5a) and after (5b) the expansion.

The corresponding tree only contains nodes which ultimately lead to an interesting system call. Moreover, the graph is expanded to a tree, so each node has a unique path to the root node (i.e., the *main* function). As a next step, the symbol names are matched against a pre-configured set to discover all resource allocations and resource accesses. A resource allocation is a node which uses a resource and may return a reference to the resource which can be used by other nodes. These nodes have a dedicated resource type and an optional access mode (e.g., extracted from the flags in the *open* call). Examples for resource allocations are the *fopen* function or the *socket* system call. Resource access nodes are accessing resources which have to be allocated somewhere else. Moreover, they may have multiple resource types they can work with. As an example, the *write* system call can not operate without a file descriptor and also works on sockets.

The last step that is needed to generate the privilege classification is the mapping from resource allocations and access nodes to privilege requirements shown in Figure 6. For each resource allocation node, one privilege requirement is created. The type of the requirement directly reflects the allocation. The access modes are gathered from the allocation and the corresponding resource access nodes.

The remaining problem is the connection of the access nodes with the proper allocation node. Our approach is based on the idea, that in most software, resource allocation and resource access is somewhat encapsulated and *near*. Data exchange between functions is done by argument passing or return values. We do not cover resource descriptors which are stored in another way (for example a global variable). However, this is part of ongoing work.

In the current implementation, the *nearest* allocation node corresponding to an access node is found by the LCA. Therefore, the following algorithm is used:

- For each allocation node, the path to the root node is labeled with the hop count to the allocation.
- The path from each access node to the root node is

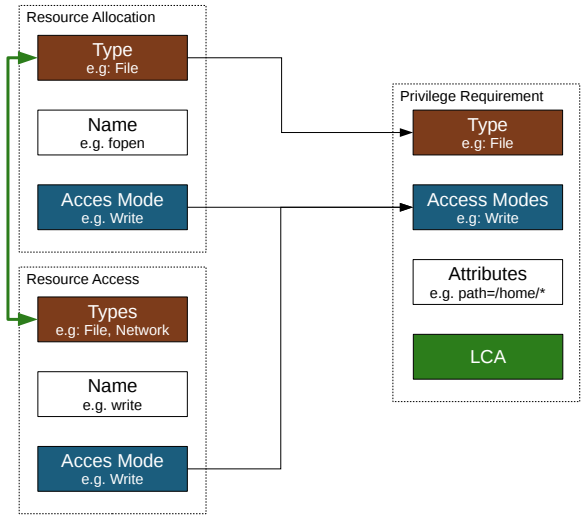


Figure 6: The generation of privilege requirements by connecting resource allocations and resource accesses.

checked for matching resource allocation labels. The node, where the sum of steps to the allocation node and steps to the access node is minimized, is the LCA.

- The access node is added to the resource allocation corresponding to the LCA.

This approach does not ensure correctness. It may happen that one access node is connected with the wrong allocation node what may lead to both, too strict and too loose policies. However, as shown in Section 5, the approach might be sufficient for many applications.

As an additional step, we delete all nodes that are neither on the path between an allocation and an access node nor on the path between a LCA and the root node. This step is only done to generate a better visualization of the results.

The result is a list of resource requirements. To complete the requirements and set the properties, the user is prompted to fill the properties for each requirement. Therefore, *PolGen* provides the LCA and the path to the root node (i.e., the stack trace) to the user. As shown in Section 5, this information often provides enough context to set the properties properly.

4. PROTOTYPE SETUP

4.1 Configuration

PolGen is implemented as proof-of-concept on Linux to check whether the system is able to generate appropriate policies and if the LCA approach is sufficient. The system is configured with the symbol names shown in Table 1.

In this setup, we set the focus on file and network I/O. The system should properly separate networking and file accesses and distinct between read/write access for files and server/client mode for sockets. Additionally, we extract the file name and access mode from *open* calls, if the parameter is a constant. Hence, the configuration covers a meaningful set of use cases but is relatively simple.

4.2 Privilege Mining and Policy Generation

On standard distributions, the non-exported function names are stripped out of the binary. This is not really a problem to *PolGen*, but the applications and some additional libraries (*libc* and *libcrypto*) have been recompiled without this step to improve the readability of the call graph.

The call graph generation is done in *C++* with the help of the *Dyninst*-library [20]. This library massively reduces the effort to build the graph and also supports somewhat platform-independent instruction parsing to get system calls and function parameters. However, in the static analysis, *Dyninst* fails to follow function pointers or virtual functions (for *C++*). While there are cases where the called function is not known at compile time, the experiments suggest that in many cases only one or a small set of possible functions can be called. At the moment, this information is added for some important function calls in the configuration but it will be automated in future work.

The remaining part of the privilege mining is implemented in Python. The resulting privilege requirements are represented as Python-objects and can be stored as *AppArmor*-policy.

5. EVALUATION

We verified the basic functionality with a simple test-program. Moreover, a real-world test has been executed with *nginx*, a common used web-server. The execution time of the privilege classification for both tests is discussed at the end of this section.

5.1 Test Program

The test program is a simple network application where a server stores the information received from the client to a file. In order to demonstrate the operation of the allocation-access-connection approach, both, the client and the server, are in the same binary and run in different threads.

PolGen is able to successfully separate the different resource parts in the test application described above. Figure 7 visualizes the internal call tree. In the figure, all function calls in the system library are hidden to improve readability. However, the library calls done in the main binary are shown with the prefix *libc*. The colored edges illustrate the connections generated by *PolGen*.

On the server-side, a socket is opened and bound. Whenever a client connects, the bytes are read and written to a file. *PolGen* correctly connects the network resource allocation (*libc_socket*) with its accesses (*libc_listen* and *libc_recv*) via their common ancestor (*server*). The connection of the file allocation and access is also done correctly. On the client-side, the program opens a connection to the server and sends some data with *libc_write*.

Table 2 shows the corresponding privilege requirements generated by *PolGen*. Only the automated classification is

Table 1: The symbol configuration of the proof-of-concept implementation.

Symbol Name[2]	Allocation/Access	Resource Type	Access Type	Additional Parameters
fopen, open	Allocation	File	Parameter(1)	file=Parameter(0)
socket	Allocation	Network		
pwrite, fwrite	Access	File	Write	
pread, fread	Access	File	Read	
write	Access	File, Network	Write	
send	Access	Network	Write	
listen	Access	Network	Server	
connect	Access	Network	Connect	

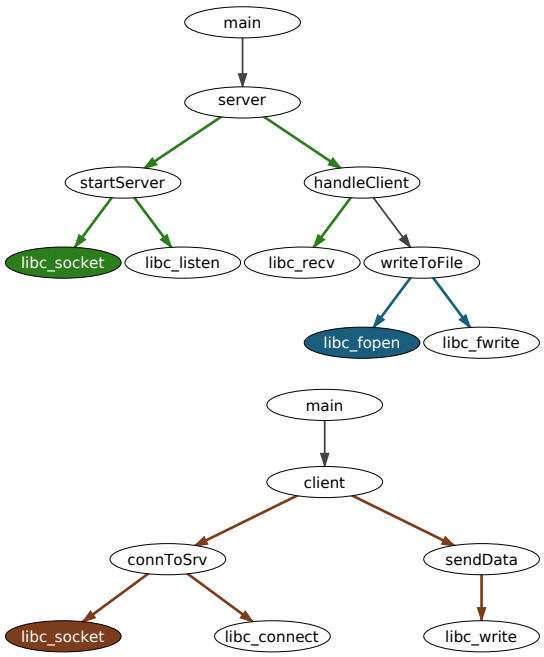


Figure 7: The anatomy of the test-program. A simple client-server application is successfully classified by *PolGen*. The program is splitted into two trees at the main node for layout reasons.

taken into account here. In a real-world scenario, the system developer would set the ports based on the system configuration and *PolGen* is able to decide whether the server needs special requirements[3].

Table 2: The resulting classification for the test program.

Resource	Access	LCA	Additional Parameters
Network	Server	server	port=
File	Write	writeToFile	file=out.txt
Network	Connect	client	port=

[2]For simplification, the source library and the distinction between library call and system call is not shown here.

[3]For example, on Linux, any application which binds to ports < 1024 needs a special capability which may or may not be allowed by the policy

5.2 NGINX

Nginx is a widely used web-server and a good candidate to examine *PolGen* with real world applications since it provides a network interface, uses configuration files and a log.

The logging functionality is a challenge for *PolGen*. Usually, a log file is opened once, propagated through the whole program, and used everywhere. This behaviour conflicts with the assumption, that resource allocation and access are close in the call tree. As a consequence, *PolGen* would connect each write to a log file to the nearest resource allocation what can lead to a too weak policy if the actual allocation is read only. However, this problem could be mitigated with techniques like data flow analysis of the log file descriptor. In the current solution, we simply blacklisted the logging functions (*ngx_log_** and *ngx_conf_log_**) as a fully automated system is not intended anyway.

Another problem results from the generation of the call-graph. As mentioned before, the static analysis fails to follow function pointers which are used in *nginx* very often to select the proper modules for a request. However, ongoing work suggests that it should be possible to statically determine most of the possible function pointers of all dynamic calls. A better call tree generation is part of a future version of *PolGen*. In the current solution, the three missing connections of the call graph for static web requests have been added to the configuration manually. Therefore, the calculated policy is not complete for *SSL* or dynamic web pages with *CGI*.

Besides these two problems, the semi-automated approach shows good results as shown in Table 3.

One resource access has been calculated fully automated. The write access to */dev/null* (4) is hard-coded and therefore *PolGen* was able to decode it.

was able to resolve the accesses to the file that stores the process-id of the main process (2)(3). The file parameters have not been auto-detected since they are set by the user. However, *PolGen* was able to detect that there are two access modes. The system developer can simply identify the appropriate file by looking at the LCA. Moreover, the LCA hints the configuration file access (8).

The two network resources (1)(6) are basically the main network socket to which the server is bound. It exists two times, because there exists a single-process, as well as a multi-process functionality in *nginx*. The port is configured

Table 3: The resulting classification for the subset of *nginx* (only considering static http requests).

	Resource	Access Mode	LCA	Additional Parameters
1	Network	Server	ngx_single_process_cycle	port=
2	File	Write	ngx_create_pidfile	file=*/var/run/nginx.pid*
3	File	Read	ngx_signal_process	file=*/var/run/nginx.pid*
4	File	Write	ngx_daemon	file=/dev/null
5	File	Write	ngx_init_cycle	file=[log]
6	Network	Server	ngx_master_process_cycle	port=
7	File	Read	ngx_open_file_wrapper	file=*/var/http/**
8	File	Read	ngx_conf_parse	file=*/etc/nginx/**
9	File	Write	ngx_reopen_files	file=[log]

Table 4: System Performance

	Functions	Function-calls	Call Generation [s]	Graph Generation [s]	Resource Classification [s]
testclient	375	1086	3.614	0.004	0.003
nginx	1137	2998	21.495	15.61	967

with the configuration file and also well known to the system developer.

For the actual server content (7), the name of the LCA does not provide any information. However, the parent path, which is also provided to the system developer, shows that the *open* function is called by *ngx_http_static_handler*. This information suggests that the public directory of the web-server is used here.

The remaining file accesses (5)(6) can not be identified statically without access to the source code. However, it is possible to use dynamic approaches.

We used modified version of *systrace* [11] to compare the static calls with the actual program execution. The web server is executed and all system calls and their stack traces are monitored. For the test run, *nginx* has been executed in multi-process mode with a minimal configuration for a *http* server. After start-up, we requested the content via a browser. Additionally, we sent all possible signals (reopen, reload, stop) to the main process. We compared the resulting list of system calls manually with the privilege requirements generated by *PolGen*.

Dynamic analysis and source code inspection showed that the remaining file accesses (5)(9) are opening the log files for writing. All other accesses to these file descriptors haven been blacklisted before. A comprehensive data flow analysis would connect the log allocations and writes, what would solve the problem. Another possibility would be a dynamic instrumentation of the program to refine the policy.

This information completes the privilege classification generated by *PolGen* and covers with the trace of system calls for the described use-case. Moreover, we compared the classification to a working *AppArmor* policy for *nginx*. The only parts which are missing in *PolGen*'s version are the entries corresponding to dynamic web pages and *SSL*, which we ignored in this test.

5.3 Performance Analysis

The classification is done on an off-the-shelf PC with an Intel Core i5-2500 ($3.3GHz$ Quad Core) and 8 GB RAM. As shown in Table 4, the test program is classified very quickly. While the number of edges and nodes of *nginx*'s call graph is only three times higher, the classification takes significant more time as for the test program. Since the test program only links against *libc* while *nginx* links against 31 libraries, the time to load and parse the application image is much higher. Therefore, it takes about ten times longer to generate *nginx*'s call tree. For the test program, *PolGen* considers only about 20 functions for the tree. All other functions are discarded because they do not lead to a system call or have been blacklisted. The tree of *nginx* contains about 750 nodes that cannot be discarded at the beginning. A big part of the consumed time could be saved by skipping the tree-expansion and allow multiple parents. Doing this in an efficient way without hiding resource accesses is part of ongoing work.

6. CONCLUSION AND FUTURE WORK

In this work, we presented a framework for automated and user-aided policy generation for application confinement. Moreover, we introduced a simple but efficient way to understand where and how resources are used in an application by finding a common caller for a resource allocation and access. A proof-of-concept implementation shows the promising results on a widely used web-server, as well as the shortcomings of the current version. For the next versions some challenges have to be taken:

The current analysis of the binary is not able to generate comprehensive call trees, especially for object oriented languages. Neither function pointers nor virtual methods can be resolved satisfactorily. In future, methods to statically detect possible function pointers or compiler extensions and source code analysis should solve these problems. Additionally, hybrid approaches with automated dynamic instrumentation of the application could be able to refine the generated classification and reduce the manual work.

Adding additional mining sources like source code, dynamic

instrumentation, existing policies or the application's functional description opens additional possibilities like verifying functional claims or generating intrusion detection systems. These possibilities are carried by a robust high level privilege classification, therefore the current specification of the privilege requirements should be refined and formalized.

The resulting tool will help developers not only to generate their confinement policies but also to understand their system and the resource requirements of the different applications running on it.

7. REFERENCES

[1] J. Salzer and M. Schroeder, "The Protection of Information in Computer Systems," in *Proceedings of the IEEE*, 1975, pp. 1278 – 1308.

[2] Y. Wu, J. Sun, Y. Liu, and J. S. Dong, "Automatically partition software into least privilege components using dynamic data dependency analysis," *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 323–333, Nov. 2013.

[3] A. Blankstein and M. Freedman, "Automating isolation and least privilege in web services," *IEEE Symposium on Security and Privacy*, 2014.

[4] S. Lachmund, "Auto-Generating Access Control Policies for Applications by Static Analysis with User Input Recognition," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, 2010, pp. 8–14.

[5] P. Centonze, R. J. Flynn, and M. Pistoia, "Combining Static and Dynamic Analysis for Automatic Identification of Precise Access-Control Policies," *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 292–303, Dec. 2007.

[6] Z. C. Schreuders, C. Payne, and T. McGill, "Techniques for Automating Policy Specification for Application-oriented Access Controls," *2011 Sixth International Conference on Availability, Reliability and Security*, pp. 266–271, Aug. 2011.

[7] Z. C. Schreuders, T. McGill, and C. Payne, "The state of the art of application restrictions and sandboxes: A survey of application-oriented access controls and their shortfalls," *Computers & Security*, vol. 32, no. 0, pp. 219–241, Feb. 2013.

[8] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," *2009 30th IEEE Symposium on Security and Privacy*, pp. 79–93, May 2009.

[9] L. Gong and M. Mueller, "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2." *USENIX Symposium on Internet Technologies and Systems*, no. December, 1997.

[10] A. Whitaker, M. Shaw, and S. Gribble, "Denali: Lightweight virtual machines for distributed and networked applications," *In Proceedings of the USENIX Annual Technical Conference*, no. Figure 1, 2002.

[11] N. Provos, "Improving Host Security with System Call Policies." *USENIX Security*, 2003.

[12] N. P. Loscocco, "Integrating flexible support for security policies into the Linux operating system," in *FREENIX Track: 2001 USENIX Annual Technical*, no. February, 2001.

[13] C. Cowan, S. Beattie, G. Kroah-Hartman, and C. Pu, "SubDomain: Parsimonious Server Security." *USENIX LISA*, no. C, pp. 1–20, 2000.

[14] Z. C. Schreuders, T. McGill, and C. Payne, "Towards Usable Application-Oriented Access Controls," *International Journal of Information Security and Privacy*, vol. 6, no. 1, pp. 57–76, 2012.

[15] A. Anurag and R. Mandar, "MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications," *Proceedings - 9th USENIX Security Symposium*, 2000.

[16] Z. C. Schreuders, "Functionality-Based Application Confinement:," Phdthesis, Murdoch University, 2012.

[17] L. Lam and T.-c. Chiueh, "Automatic extraction of accurate application-specific sandboxing policy," *Recent Advances in Intrusion Detection*, 2004.

[18] L. Koved, M. Pistoia, and A. Kershenbaum, "Access rights analysis for Java," *ACM SIGPLAN Notices*, 2002.

[19] E. Geay and M. Pistoia, "Modular string-sensitive permission analysis with demand-driven precision," in *IEEE 31st International Conference on Software Engineering*, 2009, pp. 177–187.

[20] J. Hollingsworth, B. Miller, and J. Cargille, "Dynamic program instrumentation for scalable performance tools," *Scalable High-Performance Computing Conference*, vol. 1994, no. May, 1994.