Evaluating the Effect of Improved Indirect Call Resolution on System Call Debloating

Vidya Lakshmi Rajagopalan
Stevens Institute of Technology
Hoboken, NJ, USA
vrajagop@stevens.edu

Georgios Portokalidis
IMDEA Software Institute
Madrid, Spain
georgios.portokalidis@imdea.org

Abstract

Applications use only a small set of the system calls made available by the operating system. Modifying programs to debloat or disallow unused system calls is a mitigation technique that can both reduce kernel attack surface and attacker capabilities for when an application gets compromised. To achieve this, existing systems generate a sound function call graph of the application and its dependent libraries and based on that, determine the minimum set of system calls used. Techniques that refine the call graph by determining the possible targets of indirect function calls have, in theory, the potential to also improve system-call debloating. In this paper, we evaluate the effects of state-of-the-art indirect calls refinement technique and we find that even though it improves the application call graph, it does not have any significant effect on system call policies. In contrast, we find that standard C library (`libc`) being used plays a more important role on restricting system calls. Context-sensitive and path-sensitive call graph refinement on `libc` could bring benefits to system call debloating.

CCS Concepts

• **Security and privacy → Systems security**.

Keywords

System call debloating, call graph

ACM Reference Format:
Vidya Lakshmi Rajagopalan and Georgios Portokalidis. 2024. Evaluating the Effect of Improved Indirect Call Resolution on System Call Debloating. In *Proceedings of the 2024 Workshop on Forming an Ecosystem Around Software Transformation (FEAST '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3689937.3695791

1 Introduction

Applications interact with the operating system through system calls (syscalls). There are more than 400 system calls in the latest Linux kernel. Attackers can employ techniques like control-flow hijacking to divert the control flow of a program to a system call code causing privilege escalation. Most applications only use a subset of these 400+ syscalls. System call debloating is a mitigation technique where unused system calls of an application as well as its libraries are filtered/blocked. This can minimize the kernel attack surface as well as reduce attacker capabilities. Existing system call debloating techniques like [8, 11, 18] have shown to be effective in mitigating kernel vulnerabilities, thwarting exploit payloads, among others.

Generating precise system call filters requires producing a sound call graph that includes all possible code paths that may execute. Soundness of the call graph is very critical so as to ensure that no system call which is actually required by the application is filtered. Hence, static analysis is preferred over dynamic analysis for generating the call graph and system call policies.

Generation of sound call graphs for applications can be challenging due to the presence of indirect control-flow constructs like indirect calls (icalls), indirect jumps (ijumps), etc. The general approach to resolving these indirect calls is to over-approximate the list of targets for an indirect control-flow instruction to the set of all functions whose address is taken/referenced in the program. While this approach ensures soundness, it results in a bloated control-flow graph, which in turn causes an over-approximated system call set. This reduces the effectiveness of system call debloating.

In this paper, we investigate how different indirect call target resolution techniques affect system call debloating. We also study the effect of `libc` and its different implementations on the system call filters.

2 Background

System call debloating is a technique that filters unused system calls of an application and its dependent libraries. While some works focus on filtering out unused system calls for the entire application and its libraries [8], other works focus on filtering unused system calls for different phases of the applications [11] [18]. The fundamental steps in building a system that performs system call filtering are:

(1) Determine the point in the code where system call filter has to be installed (partition boundary).
(2) Obtain the system calls reachable from the partition boundary and determine the system calls to be filtered.
(3) Enforce the system call filter at the partition boundary.

In case of SYSFILTER [8], the system call filter is placed at the beginning of the application while in case of TSP [11] and SysPart [18], it is placed at the start of the serving phase of the server. System call filters are enforced using seccomp-BPF, which is a Linux kernel facility for installing system call filters. System calls filtered using Seccomp-BPF cannot be re-enabled later in the program execution in order to prevent an attacker misusing it in case of a compromised application.

FEAST '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1233-3/24/10
https://doi.org/10.1145/3689937.3695791

One of the fundamental steps in obtaining the list of system calls to be filtered for an application is generating a sound function call graph of the application. Soundness of call graph is very important so as to ensure a correct system call filter, which does not miss any needed system call. Call graph recovery is challenging due to the presence of indirect control-flow transfers like indirect function calls, indirect jumps, virtual functions etc. The general approach for call graph recovery is to overapproximate the list of call targets for an indirect control flow to the list of functions whose address is taken/referenced anywhere in the program. This overapproximation can bring in a lot of system calls which might not be actually used in the code executed in the partition.

There have been works that focus on refining the call graph by employing pointer analysis techniques like Andersen's analysis [11] [12] or type based matching of icall sites to address-taken functions [16] [23] [21].

In this paper, we try to answer the following questions:

- How does the statically determined system call set compare against the system calls that is seen dynamically?
- Does different indirect call refinement techniques have an effect on the system calls filtered?

3 Statically Defined Filters vs. Dynamically Observed Syscalls

We first compare the statically determined system calls to the system calls observed dynamically. We evaluate the SPEC CINT2006 (SPEC) benchmark. For static analysis, we generate call graph of the application and its dependent libraries such that indirect calls (icalls) target all address-taken (AT) functions in the application and dependent libraries. We use SysPart's [18] AT algorithm for generating callgraphs and use its syscall generation algorithm, which is an improvement over Egalito's [22] `FindSyscalls()` pass to determine syscalls invoked from all functions. These results are used by a graph algorithm to compute the syscalls reachable from `main()`. For dynamic analysis, we run SPEC ref benchmark and obtain the inputs to the benchmarks, followed by which we run the benchmarks with those inputs and get the executed system calls using `strace`.

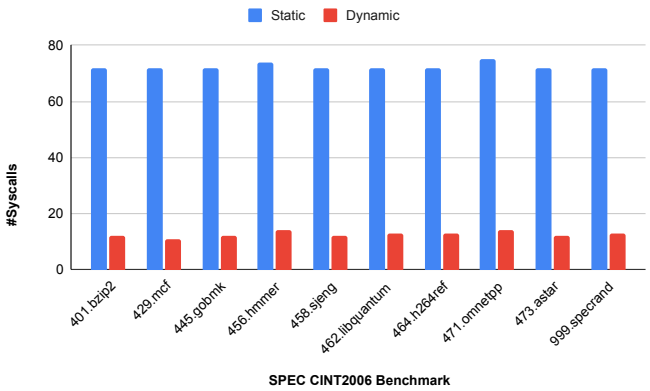

Figure 1: # Syscalls observed statically vs dynamically.

The results are shown in Figure 1. We observe that only around 20% of statically observed system calls is seen dynamically.

4 Analysis of Static Techniques

To better understand this massive difference observed in the previous section, we analyze the number of system calls statically deemed necessary under following scenarios. The scenarios we examine are listed below, while results are shown in Figure 2.

(1) **Whole call graph**: Syscalls reachable from main() with icalls of app and `glibc` call graph resolved to AT functions in entire code of application and dependent libraries.
(2) **Whole call graph with reachable AT**: Syscalls reachable from main() with icalls in app and libraries resolved to AT functions that are address-taken in functions reachable from main() (algorithm in Nibbler [3]).
(3) **Whole app call graph + direct `glibc` call graph**: Syscalls reachable from main() with icalls of app resolved to reachable AT functions while for `glibc` we only consider the direct edges.
(4) **Direct app call graph + direct `glibc` call graph**: Syscalls reachable from main() by only considering direct edges of both app and `glibc`.
(5) **Reachable from AT functions**: System calls which are reachable through direct edges from reachable AT functions.

We see a significant (approximately 44%) drop in system calls when icall targets are pruned to those AT functions which are taken in reachable functions from main(). Even with this drop, only around 31% of statically analyzed syscalls (item 2) are seen dynamically. Syscalls by including only direct edges account (item 4) for around 73.31% of total syscalls observed statically (item 2), which implies around 36.69% is contributed through indirect call edges. Syscalls reachable from AT functions (item 5) account for almost 98.83% of total syscalls (item 2).

In case of applications like servers, we observe that certain code paths are only executed if a certain configuration is enabled. Identifying configuration-specific code as well as rarely executed paths like error code requires deeper understanding of the application-specific code. Removing unused paths can be helpful in implementing effective code and syscall debloating for the applications. Additionally, reducing the AT function list and/or completely resolving icall targets could improve the call graph precision, that can further improve syscall debloating.

5 Effect of `libc`

In case of C programs, most system calls are invoked from within the C standard library. `glibc` [1] is GNU implementation of `libc` and is widely used, while `musl` [2] is a lightweight implementation of the C standard library. We compare the number of icall sites and AT functions in SPEC applications to that found in `glibc` and `musl` in Table 1. The number of icall sites is significantly high in `glibc` as compared to the applications, while `musl` has significantly lesser icall sites and AT functions than `glibc`.

We also looked at the number of syscalls reachable from commonly used C standard library functions in `glibc` and `musl` using SysPart's AT implementation. The results along with the reduction in syscalls when `glibc` is compared with `musl` are shown

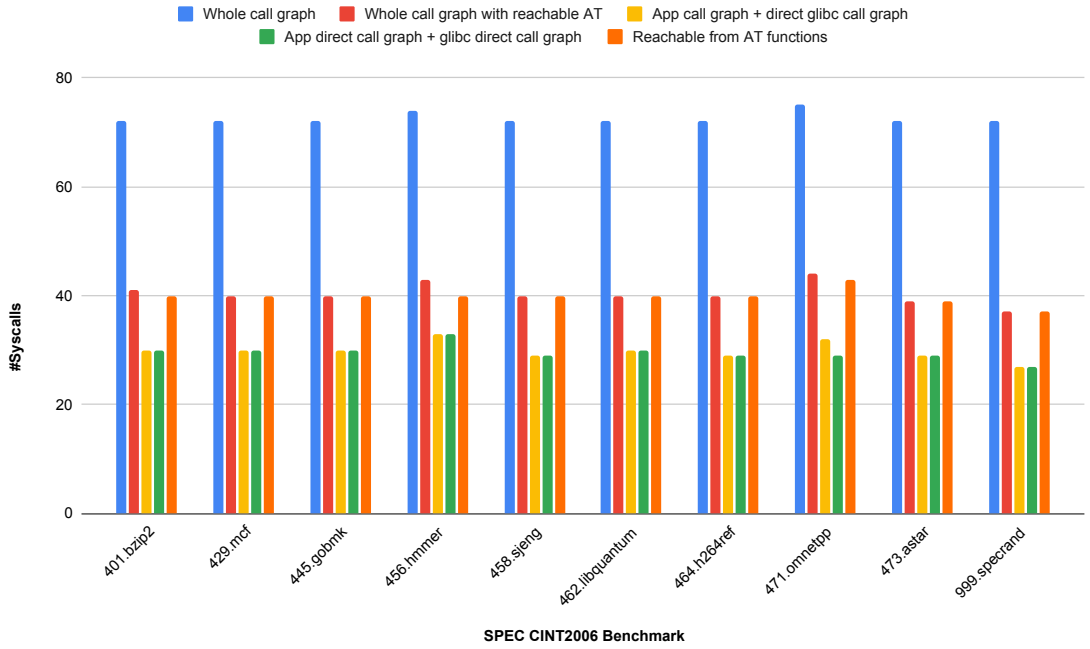

Figure 2: # Syscalls observed statically with different sets of call graphs/functions.

Table 1: #Icall sites and AT functions in different SPEC applications, `glibc` and `musl`c

Benchmark	#Icalls	#AT
401.bzip2	23	2
429.mcf	3	0
445.gobmk	47	1786
456.hmmer	14	19
458.sjeng	4	7
462.libquantum	3	1
464.h264ref	353	39
471.omnetpp	882	1016
473.astar	4	3
999.specrand	3	0
`glibc`	1080	536
`musl`c	96	73

in Table 2. `musl`c's implementation of commonly used memory allocation functions like `malloc()` and `free()` reduces the system calls by 93.05% and 97.22% when compared with that of `glibc`. Inspection of `glibc` code has shown that the callgraph of `glibc` is highly complex with a lot of cyclic dependencies between C functions. Also, `glibc` consists of condition based code as well as error handling code, which are executed very rarely. For example, we observed that in `glibc`, some syscalls are implemented to be cancellable by invoking `__pthread_enable_asynccancel()`. This function contains an indirect call in its call path and hence the size of statically determined set of syscalls for `accept4()` which is implemented to be cancellable is 72. While not considering this call path makes the syscall set size to be 1. Another example, is in case of `glibc` implementations of `malloc()` and `free()`. The call paths of these functions contain call to the function `__tunable_get_val()`, which contains an indirect call. Excluding this call path would reduce the statically computed number of system calls of these functions by 75%. Hence, identifying and understanding the context of such indirect call paths is essential to devise methods that can effectively handle them, either by resolving those indirect calls or by applying separate system call partition for those rarely executed code. This can significantly improve syscall debloating. Also, most functions that handles errors do not return and the program exits after that. Excluding non-returning paths in Nginx servers has shown to reduce the system calls of the program by 12%.

Table 2: #Syscalls reachable from `glibc` and `musl`c implementations of commonly used C standard library functions using static analysis

C function	`glibc`	`musl`c	% reduction
`accept`	72	47	34.72
`close`	72	47	34.72
`execve`	1	1	0
`exit`	72	47	34.72
`free`	72	2	97.22
`getaddrinfo`	72	51	29.16
`send`	72	47	34.72
`printf`	72	47	34.72
`malloc`	72	5	93.05

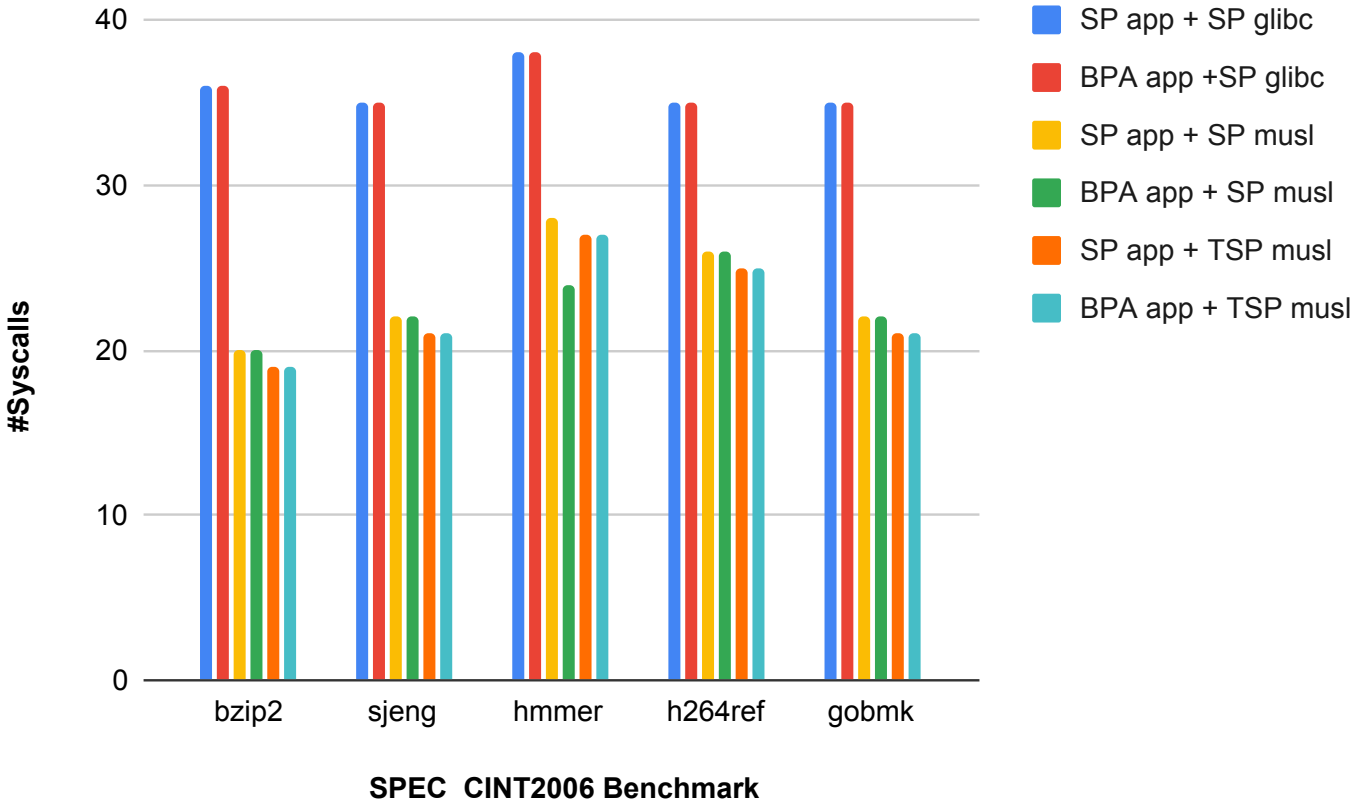

Figure 3: # Syscalls observed using different call graph refinement techniques. `libc` call graph results for `musl`c and `glibc` is shown.

6 The Effect of Icall Resolution Techniques

There have been many static analysis based techniques that aim to resolve indirect call targets of programs. They can be categorized as pointer analysis techniques and type-based matching techniques. Some of them are source-code based approaches while others work on binaries. We use some of these tools to study the effect of indirect call refining on system call debloating. The following tools were chosen for our analysis, mainly because they are publicly available and functional.

Andersen's algorithm with type based pruning (TSP): Andersen's analysis is a pointer-analysis algorithm that derives possible targets of pointers in a program. The SVF implementation of Andersen's analysis is field-sensitive but not context-sensitive or path-sensitive. In TSP [11], this graph is further refined by performing icall target matching based on types as well as including only those AT functions taken in the reachable part of the program.

BPA: BPA [14] is a binary-level points-to analysis framework that works on binaries. It performs static points-to analysis for resolving targets of indirect calls of binaries. We observed that BPA produces the icall targets of only the application and not the dependent libraries, and hence we have used it to produce call graphs of the application only.

SysPart: SysPart's call graph generation works on binaries and performs the following techniques to refine icall targets:

- Include only those AT which is taken in functions reachable from main()
- Employ use-def analysis to resolve icall targets
- Employ use-def analysis to refine AT list by removing those AT which does not flow into global variables or is returned from a function or passed as argument to a function. Also, AT functions which are only used as targets of icalls are also removed from the AT list.

We explored various call graph refinement techniques to refine `glibc` callgraph. Most source code based icall resolution techniques are LLVM/Clang based and building `glibc` with LLVM/Clang correctly is a laborious and error-prone task. Binary based call graph techniques like BPA could not handle huge shared library like `glibc`. Hence, due to the lack of a practical call graph refinement technique for glibc, we could analyze `glibc` with only SysPart.

For this experiment, we generated application call graphs of SPEC benchmark using SysPart and BPA, `musl`c call graphs using SysPart and TSP and `glibc` call graph using SP. We computed system calls with combinations of the application call graph and `libc` call graph. The results are shown in Figure 3.

We observed that there is a significant difference in system calls (around 25%-47% reduction) when `glibc` is replaced with `musl`c. Refining only the call graph of the application results in only slight reduction in system call numbers with the maximum being 16.66%

syscall reduction for `456.hmmer` when its callgraph is generated by BPA when compared to SysPart.

7 Key Take Aways

Using `muslc` instead of `glibc` can significantly reduce the number of syscalls determined statically. We also observe that the implementation of various C standard library functions in `glibc` is more complex than `muslc` and uses condition based code and error handling code. Nevertheless, the number of syscalls observed dynamically for these C functions using both implementations is similar. Hence, employing a context-sensitive and/or path-sensitive static analysis approach to handle different control flow paths in `glibc` can produce refined system call filters. Further, complex applications with more icalls and AT functions can benefit from context-sensitive analysis, if combined with a refined `glibc` call graph.

8 Related Work

System Call Filtering: There has been significant work in the field of system call filtering, where the main focus lies in the technique of generating system call policies. Static based approaches like Abhaya [17], Chestnut [7], Confine [10], and sysfilter [8] generate an over-approximated control-flow graph and generate the system calls to be filtered. Most of them are source-code based except sysfilter, which works on binaries. SIT[24] uses a combination of static and dynamic analysis to generate system call policy. BASTION [13] introduces the concepts of call type integrity, control-flow integrity and argument integrity for system calls which can be used to enforce correct use of system calls at runtime. TSP [11] introduces a temporal system-call filtering technique for server applications. It tailors different filter rules for different server phases. SysPart [18] provides an automated, binary-only, and robust temporal system-call filtering for servers. Confine [10] is a system call filtering system that limits the system calls within containerized applications. While TSP, SysPart and Confine enforce system call filters using seccomp-BPF, which can only disable system calls and doesn't allow re-enabling once disallowed system calls, SysX-CHG [9] introduces a system that allows programs to exchange filters at runtime during `execve`. This enables programs to run with a reduced set of system calls, regardless of the system calls required by its child programs which will be executed in the future.

Indirect call resolution: Call graph generation techniques have been studied for over decades now. Earlier works in call graph generation focussed on pointer analysis to determine all possible values of each memory location and thereby determine the value of function pointers used to invoke indirect call. Andersen's analysis[12] and Steengard's[20] algorithm are examples of this. Value-Set Analysis[5] and BPA[14] are points-to analysis techniques for binaries. While pointer analysis techniques are precise, they are not scalable due to the large number of memory operations that it involves [11] [19].

Function signature analysis (FSA) [4] resolves icalls by matching the types of function pointers with the ones of AT functions. FSA is scalable, but suffers from imprecision [16]. TypeArmor [21] presents an arity-based icall refining technique for binaries where icalls are matched against those AT functions by matching its arity and return information. Lin et al. [15] presents how compiler optimization impacts function signature recovery in binaries and proposes improved policies to recover function signatures for arity-based icall refining. Works like TSP [11] proposes refining the icall targets through type based matching of struct arguments. MLTA [16] and SMLTA [23] considers field types of multi-layer structures to refine icall targets. Kelp [6] proposes a hybrid approach combining type analysis with regional pointer information to generate precise icall targets in a scalable way.

9 Conclusion

In this paper, we study the effect of indirect call refining on system call debloating. We perform our analysis on SPEC benchmarks. We compare system calls computed using static analysis techniques to that observed dynamically and observe that only around 20% of statically computed system call is seen dynamically. Further, we observe that around 73% of syscalls computed statically is contributed through direct edges and syscalls contributed by AT functions is around 98.83%. Hence, deeper understanding of application code through context-sensitive and path-sensitive analysis could shed light on which paths can be excluded during syscall computation analysis that can potentially improve syscall debloating. Also, we study the effect of different standard C library implementations on the system calls. We observe that `muslc` implementations of commonly used C functions reduce the number of system calls in a significant way when compared to `glibc`. We conclude that context-sensitive static analysis combined with further refinement of icalls mainly in `glibc` can produce better system call filters.

References

[1] [n. d.]. Glibc. https://www.gnu.org/software/libc/.

[2] [n. d.]. Muslc. https://musl.libc.org/.

[3] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: Debloating Binary Shared Libraries. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (San Juan, Puerto Rico).

[4] D.C. Atkinson. 2004. Accurate call graph extraction of programs with function pointers using type signatures. In *11th Asia-Pacific Software Engineering Conference*. 326–335. https://doi.org/10.1109/APSEC.2004.16

[5] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction*, Evelyn Duesterwald (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 5–23.

[6] Yuandao Cai, Yibo Jin, and Charles Zhang. 2024. Unleashing the Power of Type-Based Call Graph Construction by Using Regional Pointer Information. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 1383–1400. https://www.usenix.org/conference/usenixsecurity24/presentation/cai-yuandao

[7] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. 2021. Automating Seccomp Filter Generation for Linux Applications. In *Proceedings of the 2021 on Cloud Computing Security Workshop* (Virtual Event, Republic of Korea) *(CCSW '21)*. Association for Computing Machinery, New York, NY, USA, 139–151. https://doi.org/10.1145/3474123.3486762

[8] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. 2020. sysfilter: Automated System Call Filtering for Commodity Software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 459–474. https://www.usenix.org/conference/raid2020/presentation/demarinis

[9] Alexander J. Gaidis, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 2023. SysXCHG: Refining Privilege with Adaptive System Call Filters. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1964âĂŞ1978. https://doi.org/10.1145/3576915.3623137

[10] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container

Attack Surface Reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 443–458.

[11] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1749–1766. https://www.usenix.org/conference/usenixsecurity20/presentation/ghavamnia

[12] Ben Hardekopf and Calvin Lin. 2007. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 290âĂŞ299. https://doi.org/10.1145/1250734.1250767

[13] Christopher Jelesnianski, Mohannad Ismail, Yeongjin Jang, Dan Williams, and Changwoo Min. 2023. Protect the System Call, Protect (Most of) the World with BASTION. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 528–541. https://doi.org/10.1145/3582016.3582066

[14] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. 2021. Refining Indirect Call Targets at the Binary Level. *Proceedings 2021 Network and Distributed System Security Symposium* (2021). https://api.semanticscholar.org/CorpusID:231835950

[15] Yan Lin and Debin Gao. 2021. When Function Signature Recovery Meets Compiler Optimization. *2021 IEEE Symposium on Security and Privacy (SP)* (2021), 36–52. https://api.semanticscholar.org/CorpusID:236340559

[16] Kangjie Lu and Hong Hu. 2019. 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* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1867âĂŞ1881. https://doi.org/10.1145/3319535.3354244

[17] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. 2020. Automated Policy Synthesis for System Call Sandboxing. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 135 (nov 2020), 26 pages. https://doi.org/10.1145/3428203

[18] Vidya Lakshmi Rajagopalan, Konstantinos Kleftogiorgos, Enes Göktas, Jun Xu, and Georgios Portokalidis. 2023. SysPart: Automated Temporal System Call Filtering for Binaries *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1979âĂŞ1993. https://doi.org/10.1145/3576915.3623207

[19] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (apr 2015), 1âĂŞ69. https://doi.org/10.1561/2500000014

[20] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 32âĂŞ41. https://doi.org/10.1145/237721.237727

[21] Victor van der Veen, Enes GÃűktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *2016 IEEE Symposium on Security and Privacy (SP)*. 934–953. https://doi.org/10.1109/SP.2016.60

[22] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 133–147. https://doi.org/10.1145/3373376.3378470

[23] Tianrou Xia, Hong Hu, and Dinghao Wu. 2024. DEEPTYPE: Refining Indirect Call Targets with Strong Multi-layer Type Analysis. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 5877–5894. https://www.usenix.org/conference/usenixsecurity24/presentation/xia

[24] Qiang Zeng, Zhi Xin, Dinghao Wu, Peng Liu, and Bing Mao. 2014. *Tailored Application-specific System Call Tables*. Technical Report.