Build Code is Still Code: Finding the Antidote for Pipeline Poisoning

Brent Pappas
brent.pappas@ucf.edu
University of Central Florida
Orlando, Florida, USA

Paul Gazzillo
paul.gazzillo@ucf.edu
University of Central Florida
Orlando, Florida, USA

Abstract

Open source C code underpins society's computing infrastructure. Decades of work has helped harden C code against attackers, but C projects do not consist of only C code. C projects also contain build system code for automating development tasks like compilation, testing, and packaging. These build systems are critcal to software supply chain security and vulnerable to being poisoned, with the XZ Utils and SolarWinds attacks being recent examples. Existing techniques try to harden software supply chains by verifying software dependencies, but such methods ignore the build system itself. Similarly, classic software security checkers only analyze and monitor program code, not build system code. Moreover, poisoned build systems can easily circumvent tools for detecting program code vulnerabilities by disabling such checks. We present development phase isolation, a novel strategy for hardening build systems against poisoning by modeling the information and behavior permissions of build automation as if it were program code. We have prototyped this approach as a tool called Foreman, which successfully detects and warns about the poisoned test files involved in the XZ Utils attack. We outline our future plans to protect against pipeline poisoning by automatically checking development phase isolation. We envision a future where build system security checkers are as prevalent as program code checkers.

CCS Concepts

• **Security and privacy → Vulnerability scanners**; *Information flow control*; • **Software and its engineering** → *Software maintenance tools.*

Keywords

Build systems, pipeline poisoning, software supply chains

ACM Reference Format:

Brent Pappas and Paul Gazzillo. 2026. Build Code is Still Code: Finding the Antidote for Pipeline Poisoning. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-NIER '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3786582.3786799

ICSE-NIER '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2425-1/2026/04
https://doi.org/10.1145/3786582.3786799

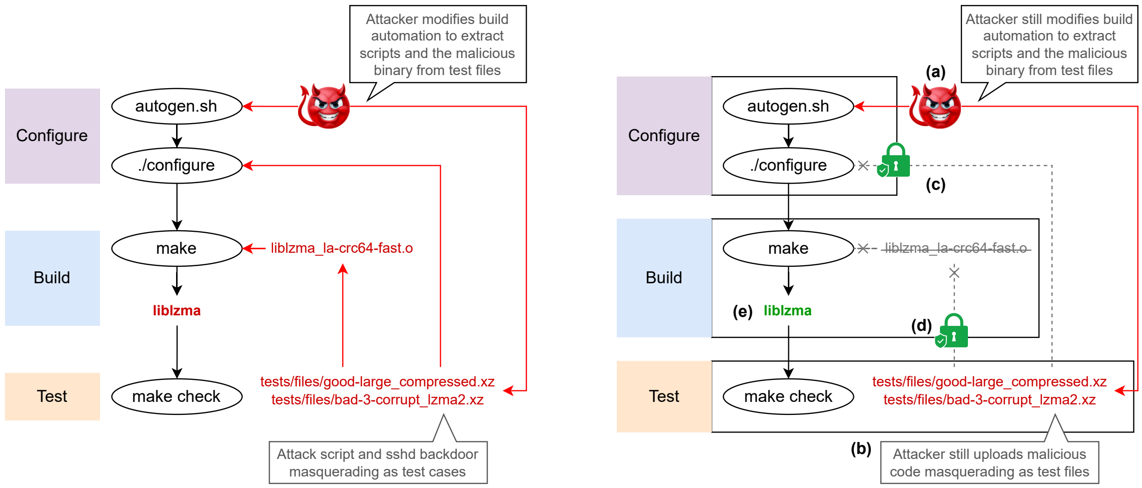
1 Introduction

Open-source C software underpins society's critical software infrastructure, including operating system kernels [2, 45, 46], secure communications tooling [36, 47], web and compute servers [1, 35, 44], and myriad other infrastructure software [7, 16, 43]. Decades of work to secure the C programming language with bug finding [5, 38, 40], program verification [3, 22, 27, 29], memory safe language translation [9, 23, 31, 39], automated testing and fuzzing [12, 41, 48], and more make implementation bugs harder for attackers to exploit. But C software is not written only in C. C codebases also include many additional languages for build automation, such as autoconf [15], Make [17], and C preprocessor code [14]. Such build code defines a C project's structure and automates its configuration, compilation, testing, packaging, and deployment [6, 15, 19].

A C project's build system is often a significant codebase with its own set of security risks. For instance, the Linux kernel's build system is over 200 thousand lines of code [34, 46]. Yet it is not only the size but the power of build automation that entails security risk. An attacker can exploit a C program without modifying a single line of C code, instead manipulating the build automation code that defines and controls how the software is constructed, tested, and released. Moreover, changes to build automation code can easily slip by developers, even when software is checked with automated bug-finding, testing, or verification tools, because build automation is not part of the program code per se. For instance, the XZ Utils Backdoor (CVE-2024-3094 [37]) hid malicious binary code in test cases and modified build code to inject the malicious code into the libzlma compression library, resulting in a backdoor to publicly-released, popular distributions of OpenSSH servers [37]. The attack illustrates *poisoned pipeline execution*, one of "OWASP's Top Ten CI/CD[1] Security Risks" [26], because it injects malicious code via the build pipeline rather than directly in C program code.

The problem is that developers do not secure build automation code as rigorously as program code despite it being vulnerable to serious attacks. We posit this problem is partly because build code is not part of the software's program code per se, only having indirect control of the behavior of the actual software. But *build code is still code.* Therefore, we should be able to secure build systems by automatically analyzing build code just as we can secure programs by analyzing program code. But analyzing build code comes with unique challenges compared with traditional program analysis. Build systems are typically written in multiple, domain-specific languages designed to produce software [8, 15, 19]. Moreover, build systems, particularly those for C systems, employ metaprogramming extensively, generating several layers of build code to produce the final build system, making analysis more difficult.

[1] Continuous Integration/Continuous Development



(a) The XZ Utils backdoor injecting itself via pipeline poisoning. **(b) Phase isolation preventing the XZ Utils backdoor.**

Even if build code analysis tools were to exist, however, build systems lack analyzable specifications of their secure behaviors. Developers use build systems to implement the software development lifecycle as discrete phases [11, 30], e.g., configure, build, test, release, etc. [13, 42, 49]. Many pipeline poisonings occur when data leaks into or out of a development phase, or when one phase illicitly accesses information belonging to another phase. For instance, the XZ Utils attack compiles code from the testing phase into the final software [7]; the SolarWinds attack edits source code during its build phase [33], and Poisoned Pipeline Execution attacks exfiltrate developer credentials during the testing phase [21]. To defend against such attacks that illicitly leak or pass information among development phases, we define a new build system security property, called *phase isolation*. A build system satisfies phase isolation when each development phase executes under a well-defined set of access permissions, i.e., the principle of least privilege. A phase isolation checker ought to catch all the above poisoning attacks.

This paper outlines our research plan for designing analyses of build automation code and checkers for phase isolation. As a proof of concept, we developed the first-of-its-kind development phase isolation checker, called Foreman. Foreman collects file access patterns of the phases of a development pipeline, checking for isolation violations against a set of phase permissions. It successfully identifies the insecure flow of malicious code from the test phase to the build phase on the compromised version of the XZ Utils codebase, demonstrating the feasibility of build code analysis to secure against pipeline poisoning. We hope our research program will kickstart an era of powerful, automated protections for development pipelines and help secure critical software supply chains against increasing attacks on build automation code.

2 Motivating example: the XZ Utils Backdoor

The XZ Utils Backdoor [37] was malicious code in the XZ Utils project that created a backdoor to SSH daemons, specifically those patched to use systemd [43], which depends on the liblzma library built by the XZ Utils repository. The attackers hid backdoor code in plain sight in the public codebase by masquerading it as a purported test case [8, 25, 32]. To obscure the linking of the backdoor code into liblzma, the attackers poisoned the build automation pipeline by modifying the code that generates the build system so that it that ultimately links the malicious code into the liblzma library.

Figure 1a shows how the build automation was poisoned. The attackers uploaded the malicious backdoor, and scripts to inject it, as test cases, since directly modifying C code would be obvious. The attackers avoided modifying the Makefiles controlling the build directly. Instead, they exploited the common autoconf build code generation process, which first generates a configuration script. The configuration script inspects system properties and dependencies and then itself generates the Makefile build system, which ultimately performs the compilation. The attackers modified an m4 macro in a file commonly included by autotools called build-to-host.m4. This macro passes commands to the shell to decompress and decrypt the first test case, which contains a script to unpack another script, which ultimately unpacks the malicious binary, called liblzma_la-crc64-fast.o. The Makefile, manipulated by the malicious scripts, then links the malicious binary with the liblzma library.

The attack illustrates a lack of isolation between phases of the build. A developer typically expects limited behavior of each phase of the build, e.g., that the compile phase touches only source code and produces a binary, while the test phase only reads test files and the resulting binary and produces a report. In this case, attackers violated this expectation by using test files, not source code or their object files, as part of the compilation phase and modified the build system indirectly by altering the code that generates it.

Figure 1b shows how building XZ Utils in a sandboxed environment enforcing phase isolation prevents the backdoor from being built into the liblzma library. Just as before, the attacker modifies the project's configuration phase (a) to inject malicious code from test case files (b) into the build phase. Now, however, a development phase isolation check (c) prevents the configuration phase from

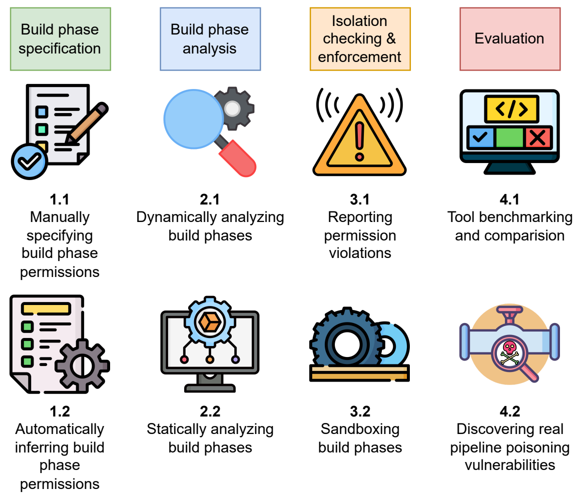


Figure 2: Development phase isolation research stages and tasks.

accessing the poisoned test files, since these files belong to the test phase and not the configure phase. Configuration then proceeds as normal, without the poisoned test files manipulating the project's Makefile to inject their contents into the liblzma library. As a redundancy, another check (d) prevents the build phase from accessing files from the test phase, making it impossible for the build phase to compile and link the contents of the poisoned test files to the liblzma binary (e). This example illustrates how phase isolation hardens build systems against poisoning, even when malicious payloads are hidden in other parts of the development pipeline.

3 Research Stages

Figure 2 divides our research plan into four stages. The first stage investigates the specification of development phase behavior. The challenge is that, while there are conventions for software development life cycles, the precise set of phases and their expected behavior lack standardization. Developers use a wide variety of automation tools and development models. Therefore, securing development phases will first require specifying phase commands and permissions. We will design methods for manually and automatically defining phases permissions. Unfortunately, it is not feasible for developers to manually specify development phase permissions for projects with large and complex build systems, so in Task 1.2 we will create tools to automatically infer phase specifications based on a project's structure and pre-existing build system standards [18].

The second stage will investigate dynamic and static analyses of phase behavior. In Task 2.1, we plan to instrument build systems to detect permission violations by tracking what information each phase accesses. Table 1 gives examples of data that development phase instrumentation could track and potential techniques for tracking permissions. Our proof-of-concept (Section 4) shows that simply tracking file read/write accesses suffices to detect the XZ Utils poisoning attack. Although dynamic analysis suffices for smaller projects, larger projects' development phases can be time-consuming and computationally expensive to execute [28]. In Task 2.2 we will explore ways to make analyzing costly builds more practical by statically predicting phase behavior with simulation.

Permissions	What is accessed	Instrumentation
Read/write	Files	Run stat() system call on files before and after running development phase
	Environment variables	Run printenv -0 before and after running development phase
Execute	Shell commands	Check command history after running development phase
	System calls	Run strace() system call before running development phase

Table 1: Example access permissions and corresponding tracking instrumentation.

In the third research stage we will develop tools using the algorithms in the previous stage to check for and enforce development phase isolation. In Task 3.1 we will create checkers to compare development phase permission specifications to actual phase behavior, and report discovered permission violations. These warnings will enable developers to catch pipeline vulnerabilities and poisonings before they reach production code. Then in Task 3.2 we will go beyond detecting permission violations, and execute development phases in sandboxed environments to prevent violations from occurring entirely. This is the method that Section 2 uses to enforce development permissions on the XZ Utils codebase and prevent the backdoor from being injected into the project's binary outputs.

We will evaluate these techniques on real-world build systems. First, in Task 4.1, we will develop a benchmark of both safe and known vulnerable applications, drawing from both real-world examples of attacks [7, 20, 24, 33] and hand-crafted poisoning attacks in existing codebases. We will use the benchmark evaluation in Task 4.2 to evaluate and improve our checkers as well as systematically check for the existence of build automation vulnerabilities in real-world code. Our review will include code from a variety of sources, including GitHub's most starred projects list, the GNU Free Software Directory, and projects considered by prior research. We will construct a standard build system review process to enable others to easily replicate our findings, and report any vulnerabilities we find to organizations such as the CVE Program to disseminate to the broader public. We will also share any strategies we find for preventing discovered vulnerabilities.

4 Preliminary Work

To show how development phase isolation can effectively prevent pipeline poisoning, we implemented a proof-of-concept checker called Foreman. Foreman wraps development phases and monitors

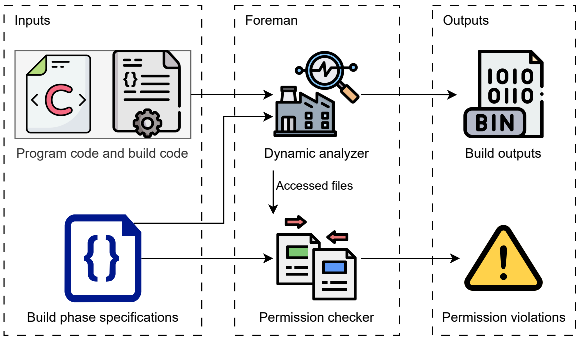


Figure 3: How Foreman checks development phase isolation.

their file accesses. Even with a simple implementation comprising only 177 lines of Python code, Foreman is already capable of detecting the pipeline poisoning form the XZ Utils backdoor.

Figure 3 is the architecture of Foreman. It takes a specification of the commands to run development phases and their expected permissions as input. Currently, we create this specification manually as a JSON file. Given the commonalities between build automation code [18], we hope to reduce the burden of specifying this via automated build specification analyses as described in Section 3.

Next, Foreman dynamically analyzes the build system by running each build phase and recording which files they access. Foreman determines if a build phase accesses a file by running the Linux `stat()` system call both before and the phase runs; if the post-phase access time is more recent, then Foreman judges the phase as having accessed the file. Finally, Foreman checks the actual file access patterns against the development phase specification. If a build phase accesses a file that the project's specification restricts that phase from accessing, then Foreman reports the file access as a permission violation by logging its details to an output file.

4.1 Case Study: Detecting the XZ Utils Backdoor

```
1 [
2  {"name":"autogen",  "commands":[["./autogen.sh"]]},
3  {"name":"configure","commands":[["./configure"]]},
4   { "name": "compile", "commands": [["make"]],
5   "restricted": ["tests/files/"] }
6 ]
```

Listing 1: JSON specification of XZ Utils build phases and file permissions.

Listing 1 is the minimal specification we defined for Foreman to detect the poisoned files. The highlighted portion on Lines 4-5 defines XZ Util's `compile` phase, which runs `make` to compile the project and is restricted from accessing files in the directory `tests/files/`, a reasonable restriction as source and test files are typically separate.

```
warning: compile accessed bad-3-corrupt_lzma2.xz
warning: compile accessed good-large_compressed.lzma
```

Listing 2: Foreman warnings about poisoned XZ Utils test files, with file paths shortened to file names.

Listing 2 shows the warnings Foreman emits after building the compromised version of XZ Utils with the build phases specified in Listing 1. Foreman warns that the `compile` phase violates its specified permissions by accessing the files `bad-3-corrupt_lzma2.xz` and `good-large_compressed.lzma`; which are the exact files that contain the malicious code used in the attack [10]. This approach shows that no specific defenses against the XZ Utils backdoor mechanism itself are needed; just by enforcing development phase isolation, Foreman detects the violation that XZ Utils backdoor depends on for the attack.

5 Future Plans

Foreman illustrates the feasibility of phase isolation checking and leads to several concrete next steps. Firstly, Foreman depends on a manual specification of phase permissions. The problem with manual specification is that if a developer incorrectly specifies a build phase's permissions, Foreman may miss vulnerabilities. We aim to mitigate this issue by taking advantage of commonalities between build phases (e.g., separate directories for source code and test code) to automatically infer build phase permissions. Additionally, an overly-strict specification can trigger false alarms. For example, in this case study we only restricted XZ Utils' `compile` phase from accessing test files, but if we had also restricted its `configure` phase from accessing test files, then we would have received dozens of false alarms, as the `configure` phase does in fact access the project's test files. Code analyses generally have trade-offs between precision and recall [4], and we plan to ameliorate these issues by exploring both static and dynamic build phase isolation checking analyses, so that developers can choose an approach that best meets their needs.

We plan to continue our work on development phase isolation checking by following the stages outlined in Section 3, beginning with the automatic inference of build system specifications. Foreman's file access tracking is coarse in that it only allows users to specify which files build phases may and may not access in any way. However, some build phases may only need permission to perform specific actions on a file like reading or writing; for instance, a compilation phase may need permission to read source files but not to write to them. We intend to extend Foreman in the future with mechanisms allowing users to specify such higher-resolution permissions, both statically and dynamically. Additionally, to make Foreman a complete development phase isolation checker, we aim to augment it with static and dynamic checks on build phase command execution permissions. Finally, we plan to make phase isolation checkers easier to use and integrate them into existing build pipelines, so that we can perform large scale evaluation of real-world build pipelines to find new vulnerabilities. While we initially focus on C software's build system, we expect the phase isolation checkers to be applicable to other build system languages by porting the analysis algorithms. Ultimately, we hope for a future where build code checkers are as ubiquitous and powerful as program code checkers.

Acknowledgments

We would like to thank to anonymous reviewers for their feedback. This work was supported in part by NSF grant CCF-1941816.

References

[1] Anonymous. OpenZFS. https://openzfs.org/wiki/Main_Page.
[2] Apple Inc. XNU. https://github.com/apple-oss-distributions/xnu.
[3] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, July 2011.
[4] Michael Buckland and Fredric Gey. The relationship between recall and precision. *Journal of the American Society for Information Science*, 45(1):12–19, 1994.
[5] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP01*, pages 73–88, October 2001.
[6] CMake. https://cmake.org.
[7] Lasse Collin. xz. https://github.com/tukaani-project/xz/, 2025.
[8] Felipe Contreras. xz backdoor and autotools insanity. https://felipec.wordpress.com/2024/04/04/xz-backdoor-and-autotools-insanity/.
[9] Correct Computation Inc. 3C. https://github.com/correctcomputation/checkedc-clang/, 2021.
[10] Russ Cox. The xz attack shell script. https://research.swtch.com/xz-script, 2024.
[11] Gerald D. Everett and Raymond McLeod Jr. *The Software Development Life Cycle*, chapter 2, pages 29–58. John Wiley & Sons, Ltd, 2007.
[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies*, WOOT, 2020.
[13] Martin Fowler. Continuous integration. https://martinfowler.com/articles/continuousIntegration.html, 2024.
[14] Free Software Foundation, Inc. C preprocessor. https://gcc.gnu.org/onlinedocs/cpp/.
[15] Free Software Foundation, Inc. GNU autoconf. https://www.gnu.org/software/autoconf/.
[16] Free Software Foundation, Inc. GNU coreutils. https://www.gnu.org/software/coreutils/coreutils.html.
[17] Free Software Foundation, Inc. GNU make. https://www.gnu.org/software/make/.
[18] Free Software Foundation, Inc. Standard Targets for Users. https://www.gnu.org/software/make/manual/html_node/Standard-Targets.html.
[19] Free Software Foundation, Inc. GNU make. https://www.gnu.org/software/make/manual/make.html, 2023.
[20] Alp Gasimov. Exclusive: 'we will see more attacks' ledger CTO warns after NPM breach.
[21] Omer Gil and AsierRF. CICD-SEC-4: Poisoned pipeline execution (PPE). https://owasp.org/www-project-top-10-ci-cd-security-risks/CICD-SEC-04-Poisoned-Pipeline-Execution, 2022.
[22] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, Savannah, GA, November 2016. USENIX Association.
[23] Immunant. C2Rust. https://github.com/immunant/c2rust?tab=readme-ov-file, 2024.
[24] John Stawinski IV. Playing with fire – how we executed a critical supply chain attack on pytorch. https://johnstawinski.com/2024/01/11/playing-with-fire-how-we-executed-a-critical-supply-chain-attack-on-pytorch/.
[25] Sam James. FAQ on the xz-utils backdoor (CVE-2024-3094). https://gist.github.com/thesamesam/223949d5a074ebc3dce9ee78baad9e27.
[26] Daniel Krivelevich and Omer Gil. OWASP top 10 CI/CD security risks. https://owasp.org/www-project-top-10-ci-cd-security-risks/.
[27] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. Static and precise detection of concurrency errors in systems code using SMT solvers. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 509–524, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
[28] Michael Larabel. Timed LLVM compilation. https://openbenchmarking.org/test/pts/build-llvm.
[29] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
[30] G. McGraw. *Software Security: Building Security in*. Addison-Wesley professional computing series. Addison-Wesley, 2006.
[31] Microsoft Research. 3C. https://github.com/Microsoft/checkedc/, 2021.
[32] Sarthak Misraa and Antonio Pirozzi. XZ utils backdoor | threat actor planned to inject further vulnerabilities. https://www.sentinelone.com/blog/xz-utils-backdoor-threat-actor-planned-to-inject-further-vulnerabilities/.
[33] U. S. Government Accountability Office. Solarwinds cyberattack demands significant federal and private-sector response (infographic). https://www.gao.gov/blog/solarwinds-cyberattack-demands-significant-federal-and-private-sector-response-infographic, 2021.
[34] Jeho Oh, Necip Fazıl Yıldıran, Julian Braha, and Paul Gazzillo. Finding broken linux configuration specifications by statically analyzing the kconfig language. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, pages 893–905, New York, NY, USA, 2021. Association for Computing Machinery.
[35] OpenSFS and EOFS. Lustre. https://www.lustre.org/.
[36] OpenSSL. OpenSSL. https://www.openssl.org/.
[37] Red Hat. CVE-2024-3094. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-3094.
[38] Thomas Reps. Program analysis via graph reachability1an abbreviated version of this paper appeared as an invited paper in the proceedings of the 1997 international symposium on logic programming [84].1. *Information and Software Technology*, 40(11):701–726, 1998.
[39] Rust Team. rust. https://github.com/rust-lang/rust, 2022.
[40] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. PhASAR: An inter-procedural static analysis framework for C/C++. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 393–410, Cham, 2019. Springer International Publishing.
[41] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*, USENIX, 2017.
[42] Peter Smith. *Software Build Systems: Principles and Experience*. O'Reilly Media, 2011.
[43] systemd. systemd. https://github.com/systemd/systemd.
[44] The Apache Software Foundation. Apache webserver. https://httpd.apache.org/.
[45] The FreeBSD Project. FreeBSD. https://www.freebsd.org/.
[46] The Linux Kernel Organization. The linux kernel. https://www.kernel.org.
[47] The OpenBSD Foundation. OpenSSH. https://www.openssh.com/.
[48] Dmitry Vyukov. syzkaller. https://github.com/google/syzkaller, 2023.
[49] Christie Wilson. *Grokking Continuous Delivery*. Manning, 2022.