Automated Synthesis of Effect Graph Policies for Microservice-Aware Stateful System Call Specialization

William Blair
Boston University
wdblair@bu.edu

Frederico Araujo
IBM Research
frederico.araujo@ibm.com

Teryl Taylor
IBM Research
terylt@ibm.com

Jiyong Jang
IBM Research
jjang@us.ibm.com

***Abstract*—We present a hybrid program analysis framework that automates the synthesis of stateful system call policies that describe admissible behaviors of containerized programs. Given a container image as input, the framework generates a reference policy that encodes a security automaton obtained by symbolically micro-executing the corresponding container's binary entrypoint under the constraints extracted from the container image metadata and environment.**

We demonstrate the utility and practicality of our approach by synthesizing security policies for 25 challenges in the DARPA Cyber Grand Challenge (CGC) corpus, 5 real-world containerized programs, including the widely used NGINX web server, and a complete microservice application from public benchmarks. We run each program or microservice using both benign and attack scenarios under the protection of a runtime policy monitor. Furthermore, we evaluate our approach by comparing our synthesized policies to those generated by four state-of-the-art system call specialization tools. Our results demonstrate that our techniques can scale to large programs and accurately extract concise reference application models for security monitoring.

***Index Terms*—system call specialization, microservices, cloud security, language-based security, binary analysis, security monitoring.**

1. Introduction

The widespread adoption of service architectures have created opportunities for compromising sensitive information and hijacking computing resources. A typical microservice architecture splits application components into multiple self-contained filesystem images (called *containers*) that an orchestration engine manages across a set of physical computing nodes. While this model streamlines service provisioning, it also significantly increases administration complexity, posing security risks that can be traced in part to the current inability of automatically refining enforceable, least privilege access policies around the deployed microservices.

Several solutions for securing microservice architectures have been proposed, including the detection of anomalous behavior in log data and metrics produced by containers [1], advanced monitoring tools to lock down container privileges [2], [3], [4], and the application of static analysis techniques to automatically infer inter-service security policies for networked applications [5]. These approaches restrict the set of system calls that an application can invoke in order to reduce the attack surface available for escaping the container abstraction. Nonetheless, they miss the opportunity to automate the creation of reference security policies around the treasure trove of information stored in container images.

A container image is a layered filesystem that contains an entrypoint program, and all the files and dependencies needed to execute the entrypoint within a sandboxed environment. Prior work has successfully collected system call sequences using instrumented compilers [6], while compiler tool chains can be used to walk the control flow graph (CFG) and learn application behaviors [7]. However, these approaches require access to source code or an intermediate representation, and statically estimating system call sequences in a CFG requires heuristics and over-approximation, since some paths in the CFG may never occur in real executions. Moreover, pure compiler analyses have a limited view of how a program interacts with its environment during execution. By contrast, dynamic approaches that learn program behavior through execution tracing often produce large models that could be condensed by using information obtained statically about the program. In addition, it can be difficult to differentiate identical system call traces produced by different program components using purely dynamic techniques.

More recently, prior work [2] has demonstrated the utility of deriving security policies through lightweight static analysis over library source code and scanning functions called by binaries. However, such an approach naturally produces stateless policies, does not consider concrete arguments passed to system calls obtained from user input or a container environment, and may miss system calls issued by dynamically generated code. In addition, prior work is primarily geared towards excluding vulnerable system calls from a workload in order to limit the possibility of kernel exploits. This prevents compromised containers from hijacking control of the kernel, but still allows container workloads to become compromised. The technique can lead to security issues when adversaries exercise system calls permitted by a stateless model to achieve a malicious goal such as those seen in mimicry attacks. For example, an attacker can issue a `connect` system call to reach out to a remote command and control server, when the `connect` is intended for a host located within the microservice. In practice, the attack is known as a server-side request forgery (SSRF) [8].

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA).

To overcome these limitations, this paper introduces μPolicyCraft, a hybrid program analysis framework that automates the generation of specialized reference security policies that efficiently track control and data flows associated with the system calls generated by containerized programs. Our key observation is that the shift to containerized environments facilitates the *introspection into well-defined units of work* from which stateful security policies can be derived with minimal human intervention and co-evolve with the microservice development lifecycle. Given a container image as input, μPolicyCraft generates a reference policy that encodes a *security automaton* obtained by statically micro-executing [9] the corresponding container's binary entrypoint under the constraints imposed by the microservice image, the application inputs, and configurations. The security automaton describes the entrypoint's states and admissible state transitions, which represent partially ordered sets of system calls guarded by constraints over their argument values. A security monitor can enforce the generated policy by tracking the observable *effects* of the microservice execution—the interactions of the program with its environment (e.g., process, network, filesystem).

Microservice architectures are best suited for our analysis because an individual microservice's computing environment is represented by an immutable container image. Each container image is represented as a layered filesystem that contains a program and all of its configuration files and runtime dependencies. The information given on an image often dictates the behavior of the program, both in terms of what system calls it will issue and the entities with which the program interacts. By micro-executing a microservice against its container image, we obtain concise and stateful security policies that are bound to a specific configuration of a program. In contrast, prior system call specialization techniques [2], [10], [4], [11], [12] employ under-constrained analyses that over-approximate a program's behavior and cause sensitive resources to be left unprotected against compromised programs.

The framework lifts binary microservice programs into an intermediate representation in which a symbolic micro execution procedure built on the Binary Analysis Platform (BAP) [13] computes an *effect graph*, a novel data structure that summarizes the program's effects guided by the constraints extracted from the container image metadata and its environment (e.g., entrypoint path and arguments, filesystem structure and configuration, environment variables). Effect graphs compactly represent individual system call sequences by highlighting program terms that invoke system calls within a lifted intermediate representation of the program. We propose a measurement of *effect coverage* to approximate effect graphs' completeness for a container image.

μPolicyCraft translates this effect graph into a security automaton, which encodes the security policy that can be deployed and enforced at the computing edge. To demonstrate the concept, we implemented a microservice-aware policy monitor (MPM) that uses a system telemetry stream [14] that encodes process provenance information and relates process events to network and filesystem activity associated with individual microservices. This telemetry stream allows the MPM to efficiently exercise the security automaton corresponding to each microservice's security policy, flagging and blocking inadmissible behaviors.

μPolicyCraft's policies can protect against mimicry attacks that are identified by malicious arguments in system calls. Other mimicry attacks may achieve malicious goals by enumerating the system calls given in our effect graphs. For example, a policy may allow an adversary to connect to a microservice, obtain data stored in memory, and send the data out using the original connection. Note that effect graphs limit the direction of communication between microservices and that such mimicry attacks would be unable to impact any resource excluded from the effect graph. As a result, an adversary cannot deviate from the effect graph to spawn reverse shells and execute commands of their own choosing. In practice, minimizing the potential impact of mimicry attacks requires analyst review (e.g., modeling adversarial capabilities to constrain effect graphs) and assistance from defenses aimed at preventing adversaries from exploiting running processes (e.g., control-flow integrity [15]).

To evaluate our approach, we synthesize policies for the challenges obtained from the DARPA Cyber Grand Challenge (CGC) corpus [16]. This corpus of applications was designed to evaluate program analysis tools' ability to automatically detect, exploit, and mitigate software vulnerabilities, and therefore serve as a good candidate and baseline to evaluate μPolicyCraft's ability to automatically synthesize and detect policy violations, since ground truth with documented security vulnerabilities is provided for all challenges. To demonstrate that our approach can scale to real-world programs, we apply μPolicyCraft to containerized binary programs, as microbenchmarks, and a complete microservice application, as a macrobenchmark. The microbenchmarks consist of 5 containerized programs (`NGINX`, a file sharing microservice implemented in Golang, `vsftpd`, `cron`, and `nullhttpd`). Furthermore, we compare μPolicyCraft to four existing system call specialization tools by comparing the tools' performance and output while modeling the `NGINX` webserver. Evaluating μPolicyCraft on microbenchmarks and comparing its capabilities to other tools allow us to confirm that μPolicyCraft can efficiently model and protect real-world programs and detect additional attack scenarios that evade existing defenses. Finally, we evaluate μPolicyCraft on a macrobenchmark by modeling and protecting a complete microservice from the DeathStarBench benchmark suite [17]. This shows that μPolicyCraft can protect real microservices in addition to individual containerized programs.

Our evaluation shows that μPolicyCraft can efficiently generate stateful security policies that embed the additional context obtained from container images to accurately model the intended control and data flow behaviors of programs that rely on the operating system's networking and filesystem. These stateful security policies can be obtained quickly, without domain expertise, and can inform runtime monitors to limit application privileges and prevent adversaries from hijacking containers. Furthermore, while purely static tools may produce stateless policies faster than μPolicyCraft, the policies produced by μPolicyCraft are bound to specific configurations of general purpose programs, and can be created in less than 40 minutes when analyzing large programs.

Contributions. Our contributions are the following:

- We present an *effect graph* abstraction and formal semantics for reasoning about program behaviors that restrict the admissible control and data effects of running containers, and a method for automatically converting these graphs into stateful security policies.
- We propose a microservice-aware policy monitor (MPM) that instantiates these policies as security automatons and efficiently tracks the effects generated by containers to detect policy violations.
- We describe our prototype implementation and evaluation, which successfully modeled 25 challenges given in the DARPA Cyber Grand Challenge (CGC) corpus, 5 real-world programs packaged as container images, and a complete microservice application from the DeathStar-Bench benchmark suite [17]. μPolicyCraft detected policy violations for all attack scenarios we present in our experiments. We have open-sourced μPolicyCraft[1].

This paper is organized as follows. We provide background for our work, including related work and our threat model (§2). We describe our system design and formalize our security policy synthesis procedure (§3). We detail our technical approach in (§4), followed by our evaluation (§5), and discussion (§6). Finally, we conclude (§7) and provide an Appendix (§A).

2. Background & Threat Model

This section discusses the key concepts used throughout the paper, including micro execution and system call specialization. We also describe related work and finish with our threat model.

Micro Execution. Micro execution [9] is a software testing technique for automatically executing binary code fragments without the need to manually define test harnesses. This capability can save significant time for testing arbitrary regions of binary programs that may be difficult to execute with traditional tools or without standing up a production environment. To execute arbitrary code fragments, a micro execution engine must accurately model a program's environment, which includes memory regions, environment variables, the filesystem, and library dependencies. For example, a micro execution engine can execute a sequence of machine instructions that dereference pointers by trapping segmentation violations and returning random data at each dereference.

Our work leverages micro execution to automatically generate a security policy that summarizes how a containerized program interacts with its environment, including how the program affects other processes, memory, the filesystem, and the network (see §3). We term these *program effects*. We argue that this approach helps alleviate the burden of manually maintaining quality security policies that a cloud operator can use to detect security violations using container telemetry. Microservices are especially suited to this analysis because each microservice container comes with an immutable layered filesystem that contains all dependencies and configuration files required to run a program. Our technique can be applied to regular binary programs, although in more general settings the program effects may need to be permissive (e.g., the `cp` program can interact with all files accessible by a user). Effects can also be tailored to a specific computing environment, such as the configuration files found on a specific server. Though our micro execution approach could improve standalone program's security, microservices are especially suitable for this analysis since every container is packaged with all files necessary to produce specialized program effects.

1. https://github.com/sysflow-telemetry/upolicycraft

Framework	Allowlisting	Temporal Order	Concrete Arguments	Container Awareness
CONFINE [2]	●	○	○	◐
sysfilter [4]	●	○	○	○
Temporal Specialization [12]	●	◐	○	○
Chestnut [10]	●	○	◐	○
Abhaya [11]	●	○	◐	○
μPolicyCraft	●	●	●	●

TABLE 1: A comparison of policy features available in recent system call specialization frameworks and μPolicyCraft.

System Call Specialization. Specializing a restricted set of system calls a program may issue is related to defining the scope of norm for application operation [18]. For example, a compact representation of the system call sequences a workload may issue can be obtained by defining n-grams, i.e., a table of finite length containing rows that express system call sequences of length n [19], [20], [21], [22], [23]. In lower level contexts, the n-gram approach has also been shown to be useful for specializing the memory access patterns for a given workload [24]. Other approaches specialize program behaviors through rule learning [25], automata generation [26], [27], Hypergraphs [28], Bayesian networks [29], Hidden Markov Models [22], [30], and neural networks [31]. Gao et al. [32] introduced the concept of an execution graph, which is built from a combination of observed system calls and concrete return addresses visited during execution. This complements static analysis that restricts system calls to individual resources in a program's environment [33], [34], [35].

More recent specialization approaches allowlist [36] system calls that can be invoked by applications by using program analysis techniques. For example, Saphire [37] performs static analysis over PHP applications combined with system call profiles of library functions found in PHP's runtime to generate system call sandboxes for web applications. Other approaches seek to restrict the network activity of Java microservices by detecting calls to network APIs [5].

System Call Specialization for Native Programs. Table 1 summarizes key attributes of recently proposed system call specialization approaches for native executables, contrasting them to μPolicyCraft. All frameworks can specialize a subset of allowable system calls (allowlisting). The temporal order attribute denotes the capability of generating policies that capture state transitions in program execution. We differentiate between approaches that identify constant system call arguments, such as constant integer values passed directly to library functions, and complete arguments that require examining the contents of structures during execution, which

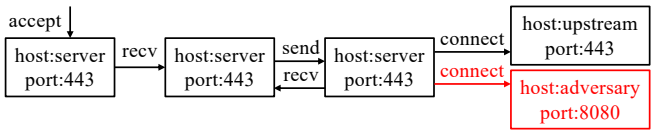


Figure 1: A segment of the security policy for a network server that proxies requests to an upstream application server detecting an SSRF attack.

μPolicyCraft does. Container awareness refers to the extent a tool uses information found in a container image's filesystem while generating a policy.

CONFINE [2] combines manually defined system call profiles of library functions, static call graph analysis, and monitoring initial container system calls to generate seccomp allowlists. Its awareness of the container image is limited to the finite monitoring window used to observe initial system calls. In contrast, μPolicyCraft examines every interaction a workload performs on a container image. Chestnut [10] refines seccomp filter generation by combining a compiler analysis to detect system calls in external libraries with binary analysis and optional binary refinements on annotated executables. Alternatively, sysfilter [4] walks the function call graph (FCG) in a native executable lifted into a binary analysis framework to inform the generation of system call allowlists as seccomp BPF filters.

Generating a system call allowlist can also be considered as a program synthesis problem. In this setting, static analysis generates a set of predicates over system call arguments that hold for the given program. Abhaya [11] uses a policy synthesizer that takes system calls and argument predicates as input and synthesizes policies to be enforced by seccomp on Linux, or Pledge on BSD operating systems, where the input predicates are restricted to constant values and arrays. Recent work attempts to divide an application between "initialization" and "server" stages to prevent adversaries from achieving mimicry attacks that execute system calls intended only for initialization [12].

Effectful System Call Policies. Fig. 1 provides a snippet of a μPolicyCraft policy that *cannot* be expressed by prior system call specialization approaches. In this security policy, μPolicyCraft restricts several system calls to specific arguments that cannot be expressed as constant values or array elements passed to the system call. For example, restricting the `accept` and `connect` system calls to a specific host and port requires examining the contents of structures passed to network system calls. Moreover, this security policy is *stateful*, and enforces a specific temporal order that is more precise than assigning lists of system calls to distinct execution phases. By temporally ordering system calls with state transitions constrained by system call arguments, μPolicyCraft policies allow security monitors to differentiate between different invocations of the same system call (different contexts). In this example, the policy allows a monitor to detect an attempted SSRF attack once an adversary tricks the workload into connecting to a malicious command and control host to install malware onto a hijacked container.

In general, such *effectful* policies can detect mimicry attacks that succeed against stateless policies by using manipulated system call arguments. Existing system call specialization techniques are primarily evaluated by their ability to restrict access to system calls that can be exploited to compromise a victim service, including the kernel and system resources. By filtering security-relevant system calls that are irrelevant to a program execution, such allowlisting approaches provide a coarse-grained reduction of attack surface. By contrast, μPolicyCraft policies model program behaviors by extracting an effect model that specializes system call sequences constrained by system call arguments that are bound to the operating environment and the context in which these system calls are invoked (such as the systems resources they affect).

In this paper, we demonstrate that micro-executing containers enables the automatic creation of accurate security policies that can be enforced over lightweight telemetry without overburdening a cloud operator's resources. The generated policies encode admissible microservice program behaviors (e.g., process control flow, filesystem, and network activity) as concise nondeterministic finite automata specialized with concrete value constraints obtained from the container image. This enables runtime security monitors to detect more subtle exploit attempts against microservices. For example, μPolicyCraft policies can identify exploits that inject malicious arguments into a legitimate system call. Our framework can therefore enhance existing security monitors, such as reference monitors that enforce mandatory access control policies and endpoint detection and response systems.

Policy Generation. Related to system call specialization are works that use safety policies to alert or enforce (sandbox) program behaviors based on a language-based security automaton [38]. Chari et al. [39] created a fine-grained policy language for limiting program capabilities from accessing certain resources, such as files. Some works operate as reference monitors (RM) inside the OS kernel [39], as a language runtime [40], within another process [41], [42], or as a combination of kernel and userspace mechanisms [43]; others are based on control-flow integrity [15]. Using stateful security policies in an RM still imposes large performance penalties compared to stateless policies that simply restrict a process to a set of system calls over finite resources on the system [39]. For this reason, state-of-the-art reference monitors available in commodity Linux systems, such as AppArmor [44] and SELinux [45], specify stateless policies for restricting the activity of protected processes. More recently, much of the research focus has been on developing both static [2], [4], [11] and dynamic [3] analysis techniques to generate seccomp [46] policies for system processes and microservices. A seccomp policy is useful for limiting the set of system calls usable by a program, yet it does not impede the misuse of the permissible set. Most closely related to *our work* is Wagner et al. [6], which used static analysis to build a call graph, and used the call graph as a security policy for comparison against a window of system calls. By contrast, our approach uses micro execution to build the policy, which takes into consideration a program's environment and parameters to build a more accurate security automaton. Furthermore, our policy monitor enforces these policies over a flow-based, entity-relational container telemetry [14], as opposed to a window of individual system calls.

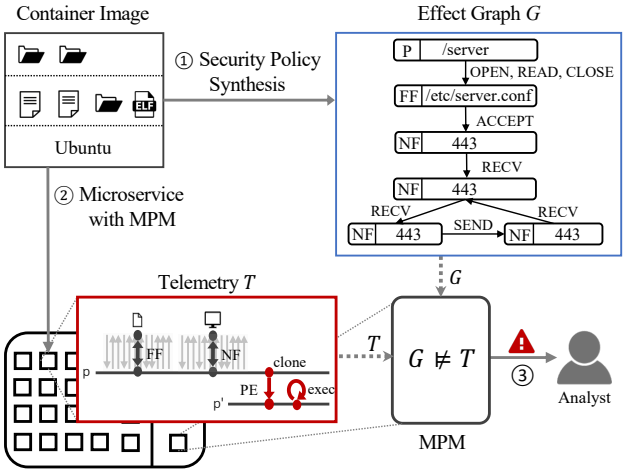


Figure 2: Architectural overview of μPolicyCraft protecting a container image by synthesizing an effect graph $\mathcal{G}$ for the image entrypoint (step ①) which a microservice-aware policy monitor (MPM) leverages to detect violations in telemetry produced by a running container (step ②).

program	$P ::= \overline{c}$
commands	$c ::= v := e \mid \textbf{store}\ e_1\ e_2 \mid \textbf{cjmp}\ e\ e_1\ e_0$
	$\mid \textbf{call}\ e \mid \textbf{ret}\ e \mid \textbf{spec}\ e \mid \textbf{halt}\ e$
expressions	$e ::= v \mid n \mid e_1 \diamond_b e_2 \mid \textbf{load}\ e$
binary ops	$\diamond_b ::=$ typical binary operators
variables	v
values	$n ::=$ values of underlying IR language
registers	r
functions	f
identifiers	$id ::=$ system call identifiers
locations	$\ell ::=$ memory addresses
prog counter	pc
effects	$\epsilon ::= \langle id, \overline{r} \rangle$
environment	$\Delta : v \rightharpoonup n$
stores	$\sigma : (\ell \rightharpoonup n) \cup (v \rightharpoonup \ell)$
function table	$\phi : f \rightharpoonup \ell$
interpreter	$\mathcal{A} : f \to \overline{r} \to \overline{\epsilon}$
call stack	$\Xi ::= nil \mid \langle f,\ pc,\ \Delta,\ \overline{r} \rangle :: \Xi$

Figure 3: A simplified intermediate representation (IR).

Program Understanding. Specification mining [47], [48] is a technique for verifying an application in order to automatically extract a formal specification of the code, which is then used by a developer to look for errors. Specification mining automates program comprehension, which can then be used to generate plausible security policies. *Program understanding* is a key goal of μPolicyCraft. Unlike specification mining, which is mostly based on runtime discovery, our approach uses micro execution to create models.

Threat Model. Our work assumes the following threat model. A cloud operator deploys container images from a continuous integration/continuous delivery (CI/CD) pipeline. Once deployed into the computing nodes managed by the cloud operator's container orchestration engine, we make the following assumptions about the containers that μPolicyCraft protects, and the capabilities of an adversary who successfully compromises a container:

- The entrypoints for the container images are binary programs implemented in systems languages such as C/C++, Go, or Rust.
- An adversary can interact with a container process running on a computing node through a network socket.
- The program contained in the image or any of its library dependencies may contain a vulnerability that, when exploited, allows an adversary to issue arbitrary system calls to the operating system kernel running the container process, which can lead to privilege escalation, information disclosure, or process control-flow hijacking [49].

3. System Overview

μPolicyCraft is a program analysis framework specifically designed to automate the generation of system call policies that can be used to monitor and harden container workloads. The hybrid program analysis symbolically micro-executes a container's entrypoint to produce a stateful security policy that a runtime monitor can use to flag non-adherent behaviors. Fig. 2 shows an overview of the framework, which comprises *offline* and *runtime* phases.

Policy Synthesis. During security policy synthesis (step ①), a container image is downloaded from a registry and the container entrypoint (bootstrap program) is analyzed by a micro execution framework with all available program parameters, environment variables, and configuration files in an offline analysis. The micro execution framework extracts a directed graph that summarizes all the interactions a well-behaved program can have with its environment, including files, networks, OS objects, and other processes. Such *effects* are distilled from the program by analysis of the system calls (including their arguments) found during micro execution. The resulting *effect graph* is synthesized into a security policy that is deployed to the microservice-aware policy monitor (MPM).

Policy Monitoring. The container image is then deployed by a container orchestration engine, which launches the container alongside the MPM in the cloud environment (step ②) during runtime. The MPM takes the security policy generated in the previous step and instantiates a finite automaton encoding the security policy. The stream of system event information T exercises the security automaton, generating alerts and optionally stopping the container when the event sequence does not match the flow of the automaton (step ③).

3.1. Security Policy Synthesis

For explanatory precision, we define μPolicyCraft's security policy synthesis procedure on a simplified intermediate representation (IR) obtained through binary program disassembling. This representation abstracts binary programs across

$$\frac{}{\sigma, \Delta \vdash n \Downarrow n}\ \textsc{Val} \qquad \frac{}{\sigma, \Delta \vdash v \Downarrow \Delta(v)}\ \textsc{Var}$$

$$\frac{\sigma, \Delta \vdash e_1 \Downarrow n_1 \quad \sigma, \Delta \vdash e_2 \Downarrow n_2}{\sigma, \Delta \vdash \diamond_b\ e_1\ e_2 \Downarrow n_1\ \diamond_b\ n_2}\ \textsc{BinOp}$$

$$\frac{\sigma, \Delta \vdash e \Downarrow n}{\sigma, \Delta \vdash \textbf{load}\ e \Downarrow \sigma(n)}\ \textsc{Load}$$

$$\frac{\sigma, \Delta \vdash e_1 \Downarrow n_1 \quad \sigma, \Delta \vdash e_2 \Downarrow n_2 \quad \sigma' = \sigma[n_1 \mapsto n_2]}{\langle \sigma, \Delta, \Xi, pc, \textbf{store}\ e_1\ e_2 \rangle \rightarrow_1 \langle \sigma', \Delta, \Xi, pc+1, P[pc+1] \rangle}\ \textsc{Store}$$

$$\frac{\sigma, \Delta \vdash e \Downarrow n \quad \Delta' = \Delta[v \mapsto n]}{\langle \sigma, \Delta, \Xi, pc, v := e \rangle \rightarrow_1 \langle \sigma, \Delta', \Xi, pc+1, P[pc+1] \rangle}\ \textsc{Assign}$$

$$\frac{\sigma, \Delta \vdash e \Downarrow n \qquad \sigma, \Delta \vdash e_{(n\,?\,1\,:\,0)} \Downarrow n'}{\langle \sigma, \Delta, \Xi, pc, \textbf{cjmp}\ e\ e_1\ e_0 \rangle \rightarrow_1 \langle \sigma, \Delta, \Xi, n', P[n'] \rangle}\ \textsc{Cond}$$

$$\frac{\sigma, \Delta \vdash e \Downarrow f \quad fr = \langle f, pc+1, \Delta, \overline{\Delta[r_1] \cdots \Delta[r_n]} \rangle}{\langle \sigma, \Delta, \Xi, pc, \textbf{call}\ e \rangle \rightarrow_1 \langle \sigma, \Delta', fr :: \Xi, \phi(f), P[\phi(f)] \rangle}\ \textsc{Call}$$

$$\frac{\sigma, \Delta \vdash e \Downarrow n \quad fr = \langle f, pc', \Delta', \bar{r} \rangle \quad \bar{\epsilon} = \mathcal{A}\ f\ \bar{r}}{\langle \sigma, \Delta, fr :: \Xi, pc, \textbf{ret}\ e \rangle \xrightarrow{\bar{\epsilon}}_1 \langle \sigma, \Delta'[r_{ret} \mapsto n], \Xi, pc', P[pc'] \rangle}\ \textsc{Ret}$$

$$\frac{\sigma, \Delta \vdash e \Downarrow id \quad \epsilon = \langle id, \overline{\Delta[r_1] \cdots \Delta[r_n]} \rangle \quad \Delta' = [\![call\ \epsilon]\!]}{\langle \sigma, \Delta, \Xi, pc, \textbf{spec}\ e \rangle \xrightarrow{\epsilon}_1 \langle \sigma, \Delta', \Xi, pc+1, P[pc+1] \rangle}\ \textsc{Spec}$$

$$\frac{\sigma, \Delta \vdash e \Downarrow n}{\langle \sigma, \Delta, \Xi, pc, \textbf{halt}\ e \rangle \rightarrow_1 \text{terminate with } n}\ \textsc{Halt}$$

Figure 4: Operational semantics of lifted programs with effects.

different instruction set architectures and models the *effects* observed by the micro execution semantics explicitly.

Intermediate Representation. Fig. 3 presents the language syntax. Programs P are represented by lists of commands, denoted $\bar{c}$. Commands consist of variable assignments, pointer-dereferencing assignments (stores), conditional branches, function invocations, function returns, system call invocations (abstracted as special invocations), and program termination statements. Expressions evaluate to typical value representations n, and comprise variables, numerical values, binary operations, and loads from memory locations. Variable names range over register identifiers, function names, and system call identifiers. We omit a formal static semantics and assume that programs are well-typed. Execution contexts are comprised of a store σ relating locations to values and variables to locations and an environment Δ mapping variables to values. Additionally, to express the semantics of effects for external function calls (e.g., runtime library API calls), we include a function table ϕ that maps external function names to their entrypoints, an interpreter context $\mathcal{A}$ that dictates whether and how each external function generates observable effects, and the call stack Ξ. The effects returned by $\mathcal{A}$ are expressed as customizable mappings from function parameters $\bar{r}$ (represented as register values) to a sequence of effects $\bar{\epsilon}$ generated by f. Effects ϵ in our analysis are defined as pairs $\langle id, \bar{r} \rangle$, where id denotes system call identifiers and $\bar{r}$ denotes the system call parameters. Note that external functions may consume files given on the container image, in addition to files that represent test inputs. See §4.1 and §5.1 for implementation details and the interface for defining test inputs.

Micro Execution Semantics. Fig. 4 presents an operational semantics defining how effects are generated by a micro-executed program. In this work, we implement μPolicyCraft against an existing micro execution framework. This semantics provides analysts a blueprint for generating an effect graph in any analysis environment that supports micro execution. Expression judgments are large-step ($\Downarrow$), while command judgments are small-step ($\rightarrow_1$). Abstract machine configurations consist of tuples $\langle \sigma, \Delta, \Xi, pc, \iota \rangle$, where pc is the program pointer and ι is the current instruction. Notation $\Delta[v \mapsto n]$ denotes function Δ with v remapped to n, and notation $P[pc]$ refers to the program instruction at address pc. We omit P from machine configurations, since it is static.

Expressions in the language are pure and programs are non-reflective. The semantics of **load** e read the value stored in memory location e. In the event that the memory location $expr$ maps to an invalid location, the micro execution framework may substitute either a fixed or random value for **load** e. This can be useful for modeling programs that rely on complex state that is irrelevant to the program's effects, such as language data structures like those found in the Go runtime. Note that this behavior enables a *probabilistic address space* to model difficult-to-test code (see §4). A *probabilistic address space* models memory regions using random number distributions. Conversely, **store** e_1 e_2 stores e_2 into location e_1. In C programs, these model pointer dereferences and dereferencing assignments, respectively. Variable assignment is a sequential instruction $v := e$ that evaluates e, updates the environment with the new mappings of v, and transitions to the next instruction $pc + 1$.

External function calls **call** f create a new stack frame fr with function arguments $\overline{\Delta[r_1] \cdots \Delta[r_n]}$, and jump to the callee's entrypoint. Returns **ret** e then consult the interpreter context $\mathcal{A}$ to appropriately collect the sequence of effects $\bar{\epsilon}$ generated by the function based on its input arguments. Context $\mathcal{A}$ can be customized and reused across micro executions to specify how effects are generated by external libraries without requiring the analysis to directly micro execute library code. This design choice allows the symbolic space to be kept small during micro execution and significantly expedites policy synthesis. Similar to **load**, the micro execution framework can substitute fixed values (e.g., a success status code like 0) for specific external function calls, random values written to return registers or relevant memory regions, or high level functions written in the analysis environment (see §4). This provides an analyst flexibility in modeling external libraries to understand the effects of a given program. Specials are commands that invoke system calls. The semantics of **spec** id generates effect $\epsilon = \langle id, \overline{\Delta[r_1] \cdots \Delta[r_n]} \rangle$, where id indexes the system call type and $\overline{\Delta[r_1] \cdots \Delta[r_n]}$ denotes the system call parameters according to the calling convention. Termination command **halt** n immediately stops the program and updates the environment with the return value n.

Effect Graph. μPolicyCraft's micro execution procedure leverages this operational semantics to produce a directed graph $\mathcal{G}$ that summarizes the admissible sequences of observable effects that a program can generate during execution. More precisely, $\mathcal{G} = \langle V, E, E \rightarrow \{(\mathrm{v}_1, \mathrm{v}_2) \mid (\mathrm{v}_1, \mathrm{v}_2) \in V^2\} \rangle$, where the pair $\langle pc, \bar{r} \rangle \in V$ is a node encoding a program state (including register values used as parameters to system calls), and $id \in E$ is a labeled edge representing an observed system call. This effect graph therefore describes a finite automaton

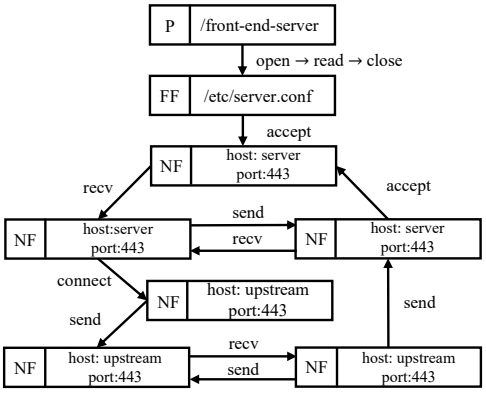
that can be used as a reference security policy for restricting permissible program behaviors. It is worth noting that we can produce effect graphs for multi-threaded programs by treating the creation of a thread as a clone operation and micro-execute thread entrypoints in parallel. See §4.1 for implementation details.

Effect Coverage. Measuring the total effects contained in a container can help an analyst determine whether a given effect graph sufficiently captures a container's behavior. The effect coverage of a given graph is defined as the fraction of the graph's effects over the total number of effects in the container's binary and library dependencies. Micro-executing every function located in external libraries allows μPolicyCraft to automatically obtain a map of functions to individual effects which are represented as system call identifiers. We micro-execute each function by exploring all paths reachable from a specific entrypoint (up to a configured path limit), ignore any memory violations, and bind every observed system call *id* to the function. This approach helps observe concrete identifiers that pass through several different registers r before reaching a system call instruction. This defines a mapping that allows us to mark the individual terms in the intermediate representation that incur effects, such as a call instruction to an effectful function. The final effect coverage therefore provides an approximate measurement of an effect graph's completeness and identifies uncovered effectful terms. This allows an analyst to quickly inspect the effects missing from their effect graph, identify terms that should be included in the effect graph, and rule out others that are irrelevant to a configuration. For example, an effect graph missing a web server's functionality as a mail server is fine if the server is configured as a TLS proxy to a backend application server.

3.2. Policy Monitoring for Microservices

The μPolicyCraft runtime instantiates $\mathcal{G}$ as a reference security policy to the MPM co-located with the protected container. The policy monitor consumes a telemetry stream that records the container activity and detects any container behaviors that deviate from the policy. The telemetry stream provides the runtime monitor with the means to exercise the security policy in lockstep with the observed system events. Any state transition not recognized by the reference policy triggers an alert and optionally halts the container.

Flow-based Monitoring. The system telemetry stream fed into the MPM is not merely a sequence of system calls, but rather a higher-level abstraction that lifts system call information into program behaviors [14]. As an entity-relational format, the telemetry has three types of objects: entities, events, and flows. Entities represent components or resources that are monitored on the system and include containers, processes, and files, while events and flows represent entity behaviors. They describe how a process interacts with resources in its environment including files, networks, and other processes. An *event* represents individual behaviors that are broken out due to their importance, their rarity, or because the order of operations is important (e.g., process clone, process exec, or file delete). Event types include process events (PE) and file events (FE). By contrast, a *flow* is a volumetric aggregation



(a)

Type	Process	Events	Entity
PE	/front-end-server	EXEC	
FF	/front-end-server	O R C	/etc/server.conf
NF	/front-end-server	A R W T	168.122.207.42:443
NF	/front-end-server	C R W T	upstream:443
NF	/front-end-server	C R W T	37.9.16.31:8080

(b)

Figure 5: A security policy for the front end server (Fig. 5a) of a microservice, and a telemetry trace that contains a policy violation (Fig. 5b).

of multiple events that fit together to describe a particular behavior. For example, all network interactions of a process and a remote host are composed of events such as `connect`, `send`, `receive`, and `close`, and can be summarized in a single bidirectional network flow (NF), while all events associated with a process's opening and reading a file can be summarized in a single file flow (FF). We chose to employ this flow-based semantics because it is more conducive to building simple (compact) yet powerful graph structures on streaming datasets that can be easily mapped to our security policies.

Vulnerable Proxy Server. To illustrate our approach, consider the public component of a microservice application, a webserver that acts as a frontend proxy to one or more microservices that run on a private subnet within a computing cloud. This server performs the important role of the microservice application's public endpoint to the Internet. In this role, the server must prevent slow or malicious clients from tying up resources to the backend application server and maintain up-to-date public key infrastructure (PKI) certificates and keys to enable secure communication with clients.

Fig. 5a shows the security policy for the frontend server as generated by the policy synthesis step. The policy describes the server first opening, reading, and then closing a configuration file, and then receiving and sending packets on port 443. After receiving a request, the server connects to the upstream server to forward the request, receive a response, and send the response back to the original client. This stateful security policy ensures that a server will only read from a file once, and thereafter only act on sockets bound to a specific port and connect to a single host specified in the policy. An adversary

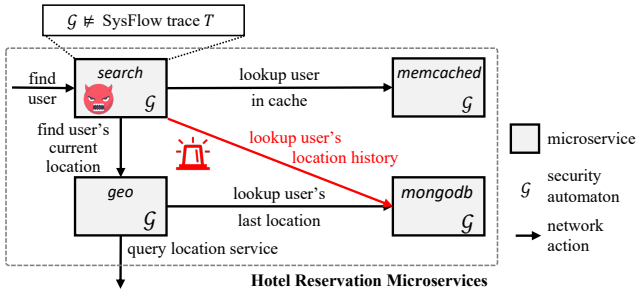


Figure 6: An MPM detecting an attempt to obtain sensitive user information through a disallowed network path within the Hotel Reservation microservice application.

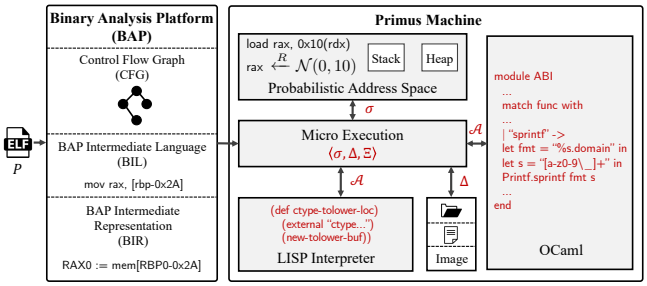


Figure 7: The different representations of a binary file lifted into BAP, and the Primus Machine (new contributions of this work are highlighted in red).

who gains access to the disk cannot disclose configuration data, since the policy forbids reading after network activity begins. Furthermore, the policy forbids an adversary from connecting to any host with the exception of the upstream server. The configuration file path, listening port, and the upstream host are automatically extracted from the program environment during our symbolic micro execution procedure. This ensures the construction of a specialized model that is tailored to the targeted microservice deployment.

When the MPM receives this security policy, it instantiates the effect graph ($\mathcal{G}$) as a finite automaton structure, with the root node of the policy assigned as the automaton's initial state. Edges in the automaton represent events or flows, while vertices represent the entities present in the telemetry in order to enable a state transition. As the MPM processes each flow-based record, it attempts to advance the automaton with the events contained in that record. A given telemetry trace (T) conforms to a security policy if the MPM can always advance the security automaton and not get stuck ($\mathcal{G} \models T$). For example, the trace in Fig. 5b shows a malicious user successfully exploiting a bug in the server and connecting to a remote command and control (C & C) server. The first record in the trace will activate the automaton (state P). The next file flow (FF) record will advance the automaton through opening (O), reading (R), and closing (C) the server's configuration file. Note that this incurs three separate state transitions in the security automaton which happen in sequence. The network record will advance the automaton through accepting (A), receiving (R), and sending (W) to a network socket opened on port 443. The next record advances the automaton through connecting (C), receiving (R), and sending (W) to the network socket made to the upstream server. The final record causes the MPM to halt, since a `CONNECT` event is only permitted to an `upstream` server. Once the MPM halts, it issues a policy violation event. The effect graph also tracks the program's user and group ids, which enables the MPM to detect privilege escalation attempts.

Vulnerable Microservice Application. Fig. 6 visualizes an attack against the Hotel Reservation application from DeathStarBench [17]. Since a single microservice's effect graph cannot express every combination of effects generated by the entire microservice application, μPolicyCraft creates a distributed effect graph composed of the effect graphs generated for each individual microservice. In addition to specializing system resources and process permissions used by each individual microservice, the distributed effect graph also encodes the valid communication patterns that occur between microservices. Our evaluation is presented in §5.4.

In this setting, each container is assigned an effect graph. An adversary who hijacks the *search* container and issues their own network requests can circumvent the *geo* container to obtain the complete history of users' locations. Since the MPM can monitor multiple microservices simultaneously, it can detect the policy violation in the *search* effect graph. The *search* effect graph forbids communication with the *geo* service's *mongodb* database. Even if an adversary successfully takes control of the database, they cannot directly connect to any microservice or external host to exfiltrate information. This is because the effect graphs and the flow-based runtime monitor track the direction of network communication (e.g., by differentiating between `ACCEPT` and `CONNECT`). To exfiltrate information from the database, an adversary has to take control of the database socket opened by the *geo* container.

4. Implementation

Our implementation consists of 3,624 lines of OCaml code that synthesizes effect graphs through micro execution and 960 lines of Primus LISP. This prototype represents our novel effect tracking plugin and supporting functionality. The MPM consists of 765 lines of Go code and is implemented as a plugin to an open-source security monitoring framework [50].

4.1. Synthesizing Security Policies

Security policies are synthesized using a new analysis plugin that we implemented within the Binary Analysis Platform (BAP) [13]. Here, we briefly introduce BAP, and describe how it enables μPolicyCraft to build effect graphs for container entrypoints.

Binary Analysis Platform. Fig. 7 visualizes how BAP processes an individual binary program. On its own, BAP is a framework for performing analysis on binaries lifted into a BAP Intermediate Language (BIL) and Intermediate Representation (BIR) that preserves the semantics of the binary's instruction set architecture and BIL operations, respectively. As shown, loading a value into the `RAX` register is accomplished by a `mov` instruction in the BIL. A special `mem` variable permits

describing loading from and storing to memory in the BIR. To lift a binary into the BIL and BIR, BAP must accurately recognize and reconstruct each function's control flow graph (CFG) via recursive descent disassembly [51], [52], [53]. The full reconstruction of program control flow graphs is generally undecidable, since one cannot establish a bijection between the set of all source programs and binary executables; however, this does not inhibit programs from recovering control flow graphs in real applications.

BIL micro execution is performed through a BAP plugin called Primus [54], which allows custom plugins to execute the BIL starting at arbitrary addresses. To automatically track the effects incurred by a given program, we implemented a new plugin that explicitly collects the effects produced by the BIR, as specified in §3.1. Our plugin tracks individual program effects by using the micro execution APIs provided by BAP and Primus, and extends Primus with advanced file and network I/O functionality to obtain accurate effect graphs. The plugin also extends Primus' string processing capability. These extensions generate a consistent view of system resources to micro-executed programs, which is critical when analyzing networked applications.

Probabilistic Address Space. Binary programs often use routines located in external libraries. During micro execution, Primus substitutes calls to external (e.g., undefined) functions with random values. This heuristic works fine for modeling programs that can execute without exact library dependencies. Furthermore, Primus allows micro execution within a *probabilistic address space*. This is useful when programs use internal state that is difficult to model, such as internal language data structures like those found within Go executables. When performing a `load` instruction, Primus will trap any memory access on an invalid (i.e., unmapped) source address and substitute a random value for memory at the address. This prevents programs from getting "stuck" due to an incomplete model of application memory. As illustrated in Fig. 7, μPolicyCraft can use a normal distribution (i.e., $\mathcal{N}(0, 10)$) as a substitute for the contents of memory given by the `rdx` register. This probabilistic address space ensures that instructions always execute, but it is insufficient to support program fragments that process strings or always expect well-formed input from byte streams.

Application Binary Interface. To accurately model external functions, we leverage the Primus LISP interpreter to model the application binary interface (ABI) of an executable. We implement LISP wrappers to abstractly interpret the effects of commonly shared library functions without micro-executing them explicitly—corresponding to context $\mathcal{A}$ in the operational semantics shown in §3.1. The level of abstraction in Primus LISP is intentionally kept close to the machine, and this enables a convenient way to alter the machine state without leaving the "familiar comforts of functional programming" [55]. This significantly expedites our analysis, as it curtails symbolic state explosion and reduces micro execution steps. For example, Fig. 7 shows a LISP stub for the `ctype` data structures that binary programs commonly use to transform characters via an array lookup. Without such functionality, system call arguments would often be excluded from our analysis. By default, BAP provides a subset of the Standard C Library implemented in Primus LISP. To support our evaluation, we improved the existing code and added our own functions. Most of the modifications and additions were related to string processing, and supporting file and network operations.

The string processing functions we modified or contributed include `strtok`, `atoi`, `getopt`, in addition to the data structures required by the `ctype` interface. All of the binaries in our evaluation heavily rely on `ctype`'s data structures to process configuration data or input. Leaving out a correct implementation of the lookup tables that support translating individual characters would limit the accuracy of our effect graphs. In addition, we contributed portions of the socket and file API. For the socket API, these are implemented enough so that networked applications properly accept our test inputs as socket data. For the file API, we provide more advanced functionality that permits introspecting files, enumerating directories, and looking ahead on file descriptors. In some cases, modeling these functions can be done by simply calling a custom Primus LISP function. In others, we use the full features of Primus LISP to perform complex input/output. This is necessary when modeling the `readv` and `writev` functions, which perform file IO by using arrays of `iovec` structs in memory.

Model Specialization. To provide micro execution with working versions of more complex APIs, we call out to the OCaml runtime through Primus LISP. This provides a feature-rich programming language for specializing program effects and providing working versions of complex functions, such as `stat`, which obtains detailed information about files in the container image. Implementing variadic functions like `sprintf` is also more convenient using OCaml, as exemplified in Fig. 7. Arguments in the socket and file API can often be represented symbolically, especially when a program uses `sprintf` to determine a hostname.

To illustrate, suppose the format argument (`fmt`) passed to `sprintf` describes a microservice's domain (i.e., `%s.domain`). Accurately fulfilling the string substitution could be a source of runtime false positives if we are unable to enumerate the list of sub-domains that `sprintf` could encounter. For this reason, μPolicyCraft cleverly substitutes the wildcard with a restricted regular expression to limit communication to any subdomain. To model servers receiving individual connections, we store the content of each network connection as a regular file in a queue consumed by `accept`. For the web servers in our evaluation, each file corresponds to a single HTTP request made to the server during micro execution. Analysts may automate ABI generation by dynamically generating return values and behaviors for external functions that are too complex to implement in Primus LISP. We have found this technique useful for modeling the Go ABI used by our evaluation microservices (see §A.1). This differentiates our approach from prior works that obtain system call policies through purely static analysis of a binary. Static techniques are unable to accurately predict system call sequences and concrete arguments generated by a binary bound to a specific container environment.

Developer Efforts. Significant effort went into modeling relevant sections of the C Standard Library and Go runtime to support our analysis. This leaves less work for analysts who can use our existing ABIs. We observed that the time to model real-world applications decreased as we refined our ABI

over increasingly complex programs. Starting on the simple `nullhttpd` server allowed us to incrementally define a C ABI used by each successive program. This greatly simplified modeling `NGINX`, which heavily used the previously defined ABIs. With the ABIs in place, modeling `NGINX` took three days worth of effort.

For modeling complex library dependencies, analysts can use the techniques previously described to model additional ABIs. Using the probabilistic address space can automatically model library data structures. Alternatively, an analyst can assign specific return codes to groups of methods. Functions that incur effects may require a partial implementation in either Primus LISP or OCaml. This provides analysts the flexibility to exclude effectless libraries (such as complicated mathematics or cryptography libraries) and focus on modeling libraries that cause interaction with system resources. The latter may often require simply adapting library arguments to existing C library stubs in our ABI.

Identifying Function Effects. We obtain the effects of external functions without having to construct the state necessary to invoke each function. This part of our Primus plugin relies on an execution style available in Primus that micro-executes all the basic blocks contained in a function in a "brute force" manner. Our plugin uses this to identify the concrete system call passed to the `syscall` instruction. System calls are identified by micro-executing each basic block reachable from the function's entrypoint (up to a maximum path length) while using the probabilistic address space. This identifies system calls that may be given in other registers or derived via some computation.

Constructing the effect graph. The core of μPolicyCraft's BAP plugin is a graph data structure called the *effect graph*, which encodes all the system call sequences observed during micro execution, along with the arguments passed to each system call. Every node v in the graph represents a specific term ID in BIR that produces effects in the form of system calls. A term ID in BAP uniquely identifies a specific location in the IR. This can be either a system call issued directly by the binary, or by calling an external library function that issues system calls, such as `bind`. Along with the term ID that issues a system call, every node v_0 also contains the concrete arguments passed to the system call obtained by examining the contents of registers. In some cases, symbolic arguments can be assigned instead, for example with `sprintf`. Storing system call arguments in nodes allows us to derive more precise security policies from the effect graph. Every edge v_1, v_2 in the effect graph is labeled by the system call issued by the node v_2. At the beginning of our analysis, the effect graph starts with a single root node that represents the process before executing the binary's `_start` routine on process startup. If μPolicyCraft observes that a library function invoked at node v_2 in the binary incurs an effect of interest, it creates an edge in the effect graph between the last visited node v_1 and v_2. Storing the unique term ID in each node allows μPolicyCraft to prevent creating duplicate nodes in the effect graph, and instead create edges that refer to previously visited nodes. This design choice leads to *more compact and generalized models*, and prevents generating overly complex security policies while micro-executing loops. At the end of the symbolic exploration, the final graph represents a subset of all the valid system call sequences a program may issue in the context of the targeted microservice deployment.

4.2. Microservice-Aware Policy Monitoring

The MPM monitors a concise flow-based telemetry stream—as opposed to a stream of individual system calls. This continuous stream of telemetry data is used by the MPM to asynchronously check adherence of container behaviors to their corresponding security policies. The `driver` captures raw system calls by passively tapping the kernel through eBPF [56]. System calls are aggregated into a flow-based and entity-relational record abstraction, and the resulting records are tracked by the MPM to enforce μPolicyCraft's policies. Processing system events using this record abstraction significantly reduces the number of system events (up to 6 orders of magnitude [14]) that are streamed and processed by the new MPM component.

Policy Monitoring with Finite Automata. When a new container starts, the MPM instantiates the effect graph obtained by micro execution into an equivalent security automaton. Like the effect graph, each state in the automaton contains the arguments for a specific system call, and the edges between states are labeled by system call identifiers. The monitor can advance the security automaton from a specific state s to s' if it observes a record with an event that matches an edge from s to s' and the entities given in the destination state match those given in the record. As an optimization, to prevent false positives for containers that prematurely exit, the monitor appends a special state s_{exit} to the container's security automaton and enable a transition from every state s to s_{exit} using an `EXIT` event. Note that the MPM can only transition to this special s_{exit} state if it encounters an exit with a normal termination code (e.g., 0).

Policy Check. To enforce the container's security policy, whenever the MPM encounters a process event PE (e.g., `CLONE`, `EXEC`, `EXIT`), it checks whether the PE is permissible from the current state in the automaton, or if the PE can initialize the automaton. If so, it advances the automaton and continues monitoring the container's telemetry. It accumulates all operations performed on file flows FF and network flows NF into a cache until the MPM encounters another PE event. Intuitively, the monitor caches sequences of file and network flows and defers checking the security automaton until a PE arrives and the workload changes, by exiting or creating a new process. Recall that the telemetry source aggregates multiple system calls into individual flows, so the size of this cache is proportional to the number of system resources (e.g., file paths, connection tuples) a workload interacts with, as opposed to the number of system calls it issues. For example, if a workload receives 1GB of data stream from a single network connection, the MPM's cache will contain a single record.

5. Evaluation

We evaluated μPolicyCraft against 25 containerized binaries from the DARPA Cyber Grand Challenge (CGC) corpus [16], 5 real-world containerized applications, and the

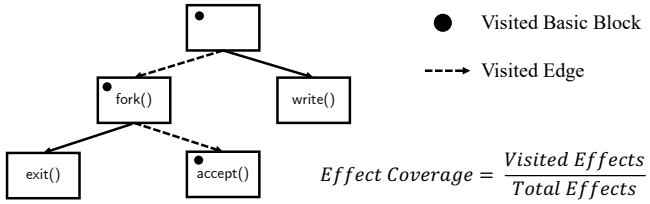


Figure 8: Micro-executing a CFG along the path visualized by the dotted line covers two out of four effects in the CFG, and achieves 50.0% effect coverage.

Hotel Reservation microservice application from DeathStarBench [17]. The CGC challenges were originally designed to evaluate the efficacy of security tools to automatically detect, exploit, and mitigate software security vulnerabilities, which makes them an ideal test set to assess the effectiveness of our approach. The higher complexity of the analyzed real-world applications shows that our approach is practical and motivates a new *coverage-guided micro execution* methodology to incrementally build security policies for the MPM. Finally, the Hotel Reservation microservice application demonstrates policy enforcement by simultaneously protecting a group of interconnected microservices. In our evaluation, the effect graphs generated by μPolicyCraft allow a policy monitor to detect a security violation due to the combination of statefulness and system call argument specialization. Furthermore, we show that effect graphs can protect programs implemented in three different languages, C/C++, Rust, and Go (DeathStarBench).

5.1. Experimental Setup

Our experimental setup consists of a single Ubuntu 22.04 LTS server with 32 Intel Xeon Gold 6130 2.10 GHz CPUs and 126GB of RAM. The server runs Docker with the MPM configured to receive container telemetry from the services considered in our evaluation. To ensure consistent results across all the artifacts in our evaluation, all binary container images analyzed by μPolicyCraft are built on top of the same Ubuntu 18.04 LTS image. The containerized binary programs were built with GCC 5.4.0. In our preliminary experiments, this was the most recent version of GCC that produced binaries for Primus to micro-execute. Details on compiler configuration are given in §5.1. The Go microservices analyzed by μPolicyCraft were built on top of the Go 1.17.3 image. As of 2023, this image has over one billion pulls on Docker Hub.

The only compiler flag we introduce for binary programs in our evaluation is `-fno-jump-tables`. This prevents the compiler from emitting jump tables. However, we have found that the latest version of BAP can recover CFGs given by jump tables. Primus can also follow indirect jumps when the destination corresponds to a valid term in the BIL, such as in an indirect function call. Debug symbols are not required. Primus can micro-execute binaries produced by newer compilers, including microservices built with a recent version of the Go compiler, which we used in our evaluation.

Coverage-Guided Micro Execution. We used our implementation of effect coverage (§3.1) to derive a test suite that exercises a program's effects when bound to a configuration in a container image. The more effects that are covered, the less likely a policy monitor using the produced effect graph will raise false positives. Coverage-guided micro execution is an iterative process where an analyst derives test inputs to cover relevant program effects for a configuration. These test inputs are represented as individual files that μPolicyCraft feeds to the program. In general, workloads may dynamically incur effects not statically captured by effect coverage. However, Primus' ability to observe effectful functions called via indirect calls enables an analyst to include these cases in their effect graphs. Fig. 8 shows an example of coverage-guided micro execution achieving 50.0% effect coverage for an individual CFG. The visibility provided by effect coverage allows an analyst to determine when a particular effect graph is complete for a configuration. For example, an analyst can see whether all the effects of a specific configuration for `NGINX` fulfill the role of a proxy server. If reaching an effectful term is difficult using test inputs alone, the analyst can directly micro-execute the routine or program fragment of interest. μPolicyCraft maintains the location of effects within the lifted program's terms. This ensures that the observed effects are assigned to the correct location in the final effect graph.

Effect coverage provides analysts with visibility into the effect terms uncovered by a graph that, if ignored, can lead to false positives in production. Furthermore, the number of effect terms is manageable for an analyst to consume. For example, only 113 terms are highlighted as effectful in `NGINX`. By iteratively defining test inputs that cover missing relevant effects, an analyst can reduce the risk of encountering false positives later on when using effect graphs to enforce security policies. Critically, covering only relevant effects can also increase true positives in production that generic allowlisting approaches may label as false negatives. For example, if an adversary exploits a container and attempts to execute effectful code in unused modules, an effect graph could catch the deviation. However, an allowlist that permits all system calls reachable in a binary would permit the attack. Furthermore, if reaching an effectful term is difficult using test inputs alone, the analyst can always directly micro-execute the routine or program fragment that contains the term. We found this to be useful for modeling Go microservices where service functionality is given within handler functions (see §5.5).

By default, the micro-executed program has full access to every file on the container image which permits access to configuration data. To define a test suite for command-line programs, an analyst can redirect Primus' `stdin` file descriptor to a file that holds the commands for the program. For the network applications included in our evaluation, μPolicyCraft refers to a directory of test cases that represent network inputs. Every time the program accepts a new connection, or asynchronously polls for events, μPolicyCraft supplies the contents of the next test case to the socket. This allows an analyst to iteratively explore new program behaviors by simply storing different network inputs within the test suite directory. Furthermore, this allows analysts to define sequences of events that trigger new effects, since test cases are applied based on the lexical ordering of their filenames. This is important for modeling asynchronous servers like `NGINX`, where an analyst needs to reply to a specific series of `epoll` events to trigger a behavior.

Coverage-guided micro execution establishes specific behavior models as goals, as opposed to an arbitrary threshold for code coverage. This is helpful for modeling containers since a complete model for a given container image will be a function of the configuration given on the image—as opposed to exercising some fraction of the entrypoint's instructions. For example, an analysis of an `NGINX` webserver configured as a proxy for a microservice can be satisfied with a final model that expresses all the behaviors of a proxy. Having visibility into the program effects covered by test inputs during micro execution enables informed decisions on constructing inputs that generate the desired model and having confidence in the final model. Contrast this with mandating specific thresholds of code coverage in generic programs which may lead to models that exhibit behavior not present for a given configuration.

5.2. CGC Challenges

Table 2 summarizes μPolicyCraft modeling the CGC programs given in our data set. Overall, μPolicyCraft can lift and model CGC programs in reasonable time frames, even on programs that perform complex numeric processing on their input, such as the FSK Demodulation program. Furthermore, μPolicyCraft is able to handle the largest programs given in the corpus. Observe that BAP's ability to lift a given executable is not necessarily dependent on program size, but rather on the diversity of program elements. For example, the CGC Hangman Game is the largest challenge in the corpus, but only because it uses large array constants to store data. For this reason, μPolicyCraft can easily model the challenge because the actual code base is quite small. In contrast, the EternalPass challenge is filled with varying functions and structures used to generate passwords, resulting in longer lifting times. μPolicyCraft is able to model the challenge nonetheless.

To select the CGC challenges for our evaluation, we modeled the challenges in numerical order while also considering the largest challenges contained in both the main corpus and challenges from the CGC qualifying event. Since all CGC challenges only use 7 system calls, they typically have similar security policies (i.e., allocate memory and transmit and receive user input). Thus, these challenges produce similar container telemetry (e.g., two file flow records). The challenges are often limited to 5k LoC when ignoring data structures stored in code. For these reasons, we chose to evaluate μPolicyCraft on more complex real-world binaries.

5.3. Microbenchmarks

Our evaluation shows that 1.) μPolicyCraft can efficiently construct effect graphs for real container workloads using image meta-data, 2.) effect graphs are representative of container behaviors, and 3.) the MPM generated from effect graphs can efficiently monitor workloads at runtime. To further demonstrate the security benefit of using the generated policies, we describe detailed attack scenarios in §5.4.

Policy Synthesis Statistics. Table 3 shows all the policy synthesis and monitoring statistics for three CGC challenges, and five real-world applications. We monitor three CGC challenges because all challenges share a similar telemetry trace. Overall, the time required to lift an application to BIL takes a few seconds on average and grows with the size of the binary. The time needed to micro-execute a challenge varied and increased for challenges that perform significant computation on their input. For example, the *FSK Demodulation* challenge must reconstruct an entire window of analog radio transmissions before it renders an output, and for this reason, it can take thirty minutes to produce its effect graph.

Service	Size	Static Analysis		
	LOC	L (s)	ME (m)	PS (# states)
Palindrome Finder	67	1.71	0.07	4
Pseudo Random Number Generator	314	1.98	13.72	5
Command Line Games	368	2.16	0.05	15
Ti ARM Emulator	470	1.97	0.36	7
WhackJack CLI Game	512	2.27	0.06	14
Shortest Path Analysis	545	2.00	2.16	21
Network File System	549	3.11	0.42	6
Timecard Management System	603	2.19	2.94	4
SCRUM Database	667	2.24	0.48	15
FSK Demodulation	788	1.26	30.48	6
GPS Package Tracker	900	2.28	1.56	4
Flight Planner	900	2.84	0.47	21
16-bit Virtual System	1,206	3.07	1.08	4
PowerPlant Control System	1,227	3.33	0.01	49
Command Line Shell	1,253	3.16	0.30	27
Embedded Thermal Controller	1,400	3.01	0.24	8
Fitness Tracker	1,430	2.75	0.06	10
Monster Role Playing Game	1,925	3.54	1.18	35
Automotive Network	2,110	3.57	40.20	6
Network Database File System	3,079	4.82	0.29	5
Planet Markup Language Parser	5,363	9.02	4.93	27
Cable Grind	34.9k	60.10	5.08	8
Confident Authentication Terminal	34.4k	37.99	2.86	6
EternalPass	322k	1801.00	21.40	5
CGC Hangman Game	416k	1.79	0.16	23

TABLE 2: Security policy synthesis statistics for DARPA CGC corpus included in our evaluation (L=Lifting time, ME=Micro Execution time, PS=Policy Size).

Policy Monitoring. Table 3 also summarizes the effect coverage achieved during policy synthesis as well as, the overall performance of μPolicyCraft. Observe that, while an effect coverage of 27% for `NGINX` may appear low, this effect graph sufficiently covers the functionality required of a proxy server. In this setting, we restrict the general purpose `NGINX` web server to a subset of its full functionality and, as a result, obtain a restricted model that only applies to a proxy's duties, as opposed to covering all of `NGINX`'s functionality which could lead to security problems. For example, if the proxy server is permitted to `exec` while handling a request, an adversary could use this to hijack control of the server. Furthermore, the size of an effect graph is proportional to the number of terms in the program that incur effects, such as a direct call to `writev`. This leads to compact models for large programs like `NGINX` since effectful terms are often located within reusable functions. Note that repeatedly enumerating existing paths in an effect graph keeps the size of the graph the same. New states are only added when new effectful terms are encountered. This helps keep effect graphs compact independent of the number of system call sequences a program may generate.

Container		Size	Static Analysis				Policy Monitoring	
Entrypoint	**Pulls**	**Binary (KB)**	**Effect Coverage**	**L (s)**	**ME (m)**	**PS (# states)**	**Processing Time (ms)**	**Transitions**
Ti ARM Emulator†	-	24	80%	1.97	4.95	7	1.50	3
FSK Demodulation†	-	36	60%	1.26	30.48	6	2.96	3
Network File System†	-	44	60%	3.11	1.26	6	2.07	3
nullhttpd	-	48	88%	1.86	3.83	19	12.85	23
cron	2M+	60	73%	6.36	0.50	29	37.79	28
file sharing	-	60	90%	1.61	0.20	70	30.33	52
vsftpd	9M+	192	46%	18.96	34.00	15	12.85	8
NGINX	1B+	944	27%	69.50	39.85	27	14.23	11

† DARPA CGC corpus

TABLE 3: Evaluation statistics for protecting workloads (L=Lifting time, ME=Micro Execution time, PS=Policy Size).

Container Image		Concretized Variables	
Entrypoint	**Symbolic Variables**	**Baseline**	**Metadata**
nullhttpd	922	172	595 (+3.40 ×)
cron	1,344	44	611 (+13.9 ×)
file sharing	847	30	467 (+15.5 ×)
vsftpd	5,159	618	2,567 (+4.20 ×)
NGINX	37,839	875	11,022 (+12.6 ×)

TABLE 4: Measuring concretization of symbolic variables in container images using image metadata against a baseline that consists of an empty environment (gain is parenthesized).

To measure the performance of μPolicyCraft, we profiled the handler within the MPM responsible for checking individual records that appear in the telemetry stream. We measured the amount of time the MPM takes to verify a single execution trace produced by each attacked program over 100 trials in order to understand the variability of the MPM's runtime due to external factors, such as the Go runtime and variations in the operating environment. Overall, we observed time spent processing records stays low. Furthermore, the individual traces transitioned each program's effect graph without incurring false positives. Next, we detail specific attack scenarios and MPM performance characteristics.

Environmental Impact on Policy Creation. To evaluate μPolicyCraft's ability to protect programs by considering both network inputs and environmental information stored in a container image, we apply μPolicyCraft to 5 real-world containerized applications: nullhttpd, cron, vsftp, `NGINX`, and a Go-based filesharing application. Table 4 empirically demonstrates the utility of relying on container image metadata to generate security policies. In this experiment, we measured the number of symbolic variables found in a container entrypoint, which is given by the number of variables found in the BIR, the program's lifted static single assignment (SSA) form [57]. We then count the number of symbolic variables that are made concrete (i.e., assigned a value) during micro execution, both on a baseline program that runs without any environment and with a program that has access to the container's files. These results demonstrate that image metadata allows μPolicyCraft to concretize substantially more symbolic variables than simply using an empty environment as a baseline. This shows that the additional information obtained from the environment enables μPolicyCraft to explore more of a program's state space and construct a more precise effect graph.

System	Modeling Time (M)	Performance Overhead
Chestnut	0.12	+0.87%
sysfilter	0.16	+1.99%
CONFINE	2.93	+1.05%
Temporal Spec.	21.36	+2.14%
μPolicyCraft	39.85	+2.96%

TABLE 5: A comparison of modeling and monitoring with different system call specialization tools.

MPM Performance. To assess the performance characteristics of the MPM, we used Apache Benchmark (`ab`) to stream a heavy workload of 1 million requests submitted by 10 simultaneous clients against the monitored proxy server for the file sharing microservice. We perform three distinct measurements to understand the performance overhead imposed by the MPM. First, we measure the impact the MPM imposes on the web server's response times. To do so, we perform our stress test on an unprotected `NGINX` instance that acts as a baseline and an instance of `NGINX` monitored by the MPM. Second, we monitor each web server's resource consumption during the stress test in order to identify whether the MPM negatively impacts the server's resource consumption. Finally, we monitor the resource consumption of the MPM to ensure the workload's normal operation.

Overall, we observed that the security monitor increases the server's response time by (+2.96%), but the response time for 99% of requests remains the same at 11ms. In both settings, the `NGINX` server's resource utilization stays consistent, with memory usage staying flat and CPU activity showing that the four workers and one management process can complete their tasks often by utilizing less than 5 CPU cores. Resource consumption stays consistent with respect to CPU and memory utilization. The MPM processor consumed up to 40MB of RAM and used at most 2 CPU cores throughout the test. Increasing the number of effect graphs held by an MPM does not drastically increase resource consumption since processing runs sequentially on each telemetry record.

Tool Comparison. To assess μPolicyCraft's ability to generate accurate security policies, we generated security policies for an `NGINX` container using four tools outlined in Section 2. We excluded the Abhaya tool from the comparison since its source code is not publicly available. We used each tool to produce a security policy for `NGINX`, and repeated the MPM performance evaluation while using seccomp to enforce the policy. Table 5 summarizes the time required to create a

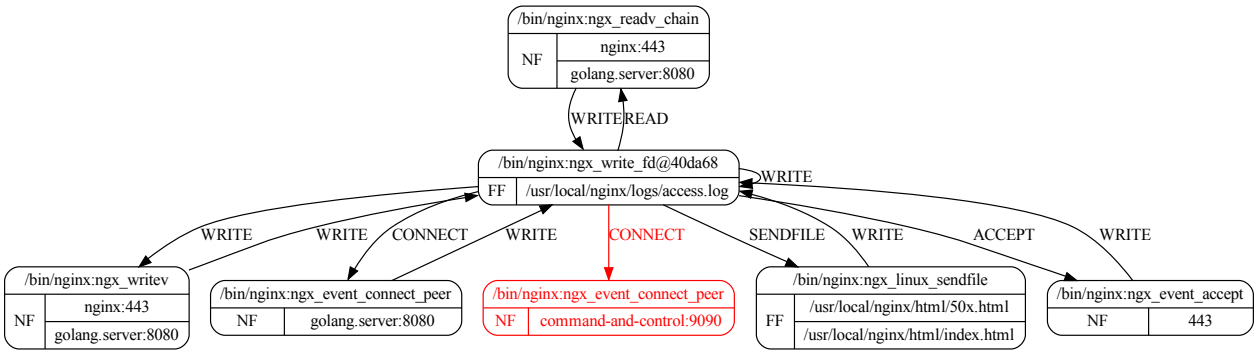


Figure 9: Detecting a server-side request forgery (SSRF) attack against an NGINX web server.

security policy for the NGINX container, and compares the performance overhead required to enforce the policy produced by each tool. Overall, all the tools came to the consensus that NGINX accepts and creates connections with remote clients, processes socket events using the epoll system calls, and may issue an execve system call. However, μPolicyCraft emits the most restrictive policy that is tailored to a specific NGINX configuration, as opposed to all possible configurations supported by the binary. These stateful policies enable the MPM to restrict the connect system call to the single host present in the configuration, as opposed to any network entity. Furthermore, μPolicyCraft only permits a single clone and execve when the container's entrypoint starts the NGINX server. Further clones are allowed by the other tools but forbidden by μPolicyCraft after the master NGINX process starts its worker processes.

5.4. Microservice Attack Scenarios

We present three attack scenarios that demonstrate μPolicyCraft's ability to synthesize security policies for non-trivial binary programs, and detect runtime policy violations. First, we describe an attack against a simple Rust microservice. Next, we present attacks against a tiered microservice composed of the backend file sharing service and the frontend NGINX web server. The MPM monitors this complete microservice stack using the security policies synthesized for each container.

While each of the attacks used in this evaluation may be detected by orthogonal security measures, we argue that the effect graphs synthesized by μPolicyCraft allow operators to restrict a container's activity without requiring domain knowledge for a particular container, and the automated security policy creation can co-evolve with the development and security operations lifecycle.

Rust Microservice. An effect graph for a Rust microservice is visualized in Fig. 10. This shows the effect graph generated for a simple Rust microservice built with the standard library's std::net module using the standard rustc compiler (version 1.68.2). This microservice can be exploited by a malicious request. The compromised microservice may change the process's effective user ID by issuing a SETUID system call. An MPM enforcing this effect graph observes the policy violation and executes a protective action.

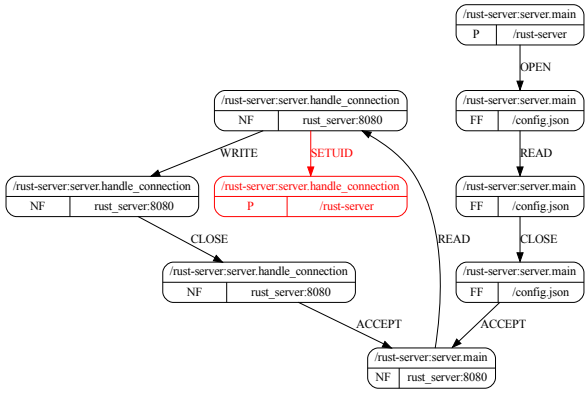


Figure 10: Security policy violation in a Rust microservice implemented using the std::net module and compiled with rustc version 1.68.2.

File Sharing Service. The Go-based file-sharing microservice used in our evaluation allows users to store and share files within managed cloud object storage. In our specific scenario, μPolicyCraft detects an exploit against the microservice that attempts to EXEC a shell on the container. At runtime, the MPM observes that the service's effect graph policy forbids the EXEC operation, and responds by raising a policy violation. This demonstrates μPolicyCraft's capability to model and protect a modern microservice.

NGINX Web Server. NGINX is a widely used web server and powers 33% of web sites found on the public Internet [58]. Fig. 9 depicts a subset of the security policy μPolicyCraft generated for a container image containing NGINX version 1.20 that serves documents from a specific directory on the filesystem and acts as a reverse proxy for a microservice. In this example, an adversary who successfully gains access to the web server can implement a server-side request forgery

(SSRF) attack, a top-10 OWASP security risk [59]. This allows the remote client to trick the web server into connecting to attacker controlled endpoints to disclose sensitive information or install malicious software. In our evaluation, we use an effect graph obtained from an `NGINX` container image to detect an attempted SSRF attack with the MPM. When an adversary attempts to connect to a command and control (C & C) server, the security automaton within the MPM will not recognize the malicious endpoint given in the telemetry trace. In contrast to system call allowlisting approaches like CONFINE [2], μPolicyCraft's policies can differentiate between individual contexts (i.e., calling program states) that issue the `CONNECT` system call.

5.5. DeathStarBench Benchmark

We evaluate μPolicyCraft on a complete Hotel Reservation microservice application from DeathStarBench [17], which contains eight containers as shown in Table 6. We perform coverage-guided micro execution on each microservice within the microservice application. We then load the microservices' effect graphs into the MPM, which monitors the entire application for policy violations while running the DeathStar benchmarks.

Micro Execution Performance. Table 6 breaks down the performance of both modeling each individual microservice and enforcing effect graphs on the application as a whole. In this setting, the MPM uses container telemetry produced by all of the application's microservices to detect attempts to disclose or alter system resources, or attempts to communicate with external services or co-located microservices. These external requests are disallowed by the security automaton. Results indicate that μPolicyCraft can model non-trivial programs in microservice applications that have large images and many library dependencies.

Attack Detection. Recall the example from §3 where a compromised *search* service attempts to disclose data from the *geo* service's database. Since the security automaton produced by μPolicyCraft forbids communication between *search* and *geo*'s *mongodb* instance, the MPM observes the policy violation and raises an alert to an operator. Since *mongodb* and *memcached* are isolated in the Hotel Reservation application, we assign them a blanket effect graph that disallows any network egress outside of fulfilling requests permitted by other services' effect graphs. Fig. 11 shows the detection of this attack on the *frontend* service's effect graph.

Each path within the effect graph represents the sequence of effects incurred by an endpoint within the microservice. For example, if an adversary creates a reservation by issuing a request to the *frontend* microservice, the runtime monitor will cause the MPM to transition the effect graph from the `Run` node to the path beginning with the `CheckUser` node. If the adversary succeeds in exploiting a bug, they may attempt to connect to a *mongodb* container co-located with the microservice to extract sensitive user data. The monitoring event for the malicious `CONNECT` operation causes the MPM to get "stuck" since a `CONNECT` to the *mongodb* container is absent from the effect graph.

Model Validation. Our proposed coverage-guided micro execution technique is designed to produce models relevant

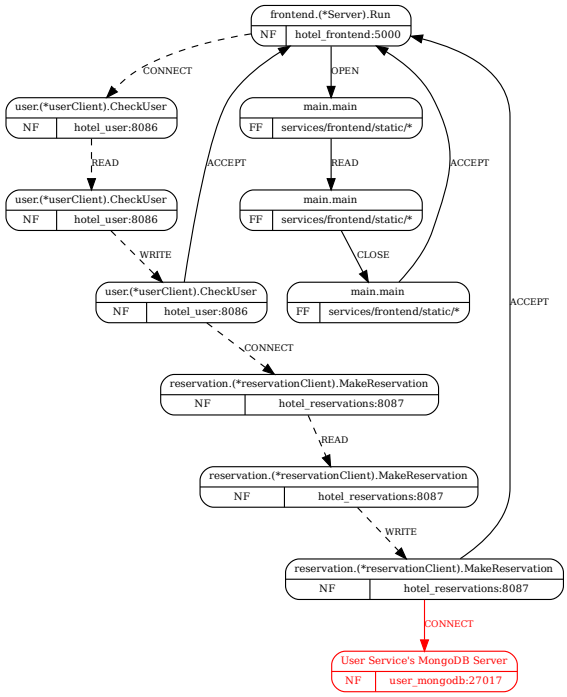


Figure 11: Detecting a security policy violation in the Hotel Reservation's *frontend* microservice. The dotted line visualizes the path taken by the MPM while consuming *frontend*'s SysFlow telemetry before encountering the violation.

to specific application configurations. Providing a model's effect coverage to an analyst allows them to decide whether an effect graph is sufficiently "complete" for a service's configuration. To measure the frequency of false positives while enforcing effect graphs with an MPM, we perform the following procedure. The Hotel Reservation benchmark suite contains a suite of test inputs to the microservice frontend that exercise the services' functionality. For example, searching for recommendations causes the *frontend* service to make API requests to the *recommendation* service, which generates additional requests.

To test our model's efficacy, we use time series split cross-validation [60] for effect graph training and validation. This technique divides the requests contained in the test suite into k individual segments. Requests made to a microservice application are naturally sequential, since each request may either refer to a previously created entity or define a new one. For this reason, we use the first $k-1$ segments to generate effect graphs, and the final k segment for validation (i.e., to measure for false positives). Other cross-validation approaches that mix the sequence of requests would likely produce lower-quality data for modeling if out-of-order requests refer to non-existing entities.

To collect the first $k-1$ segments, we bring up a live version of the Hotel Reservation application and collect all the inputs to its microservices by collecting network traffic with the `tcpdump` utility. Once we have obtained all the

Hotel Reservation	Size	Static Analysis				Policy Monitoring			
Microservice	**Binary (KB)**	**Effect Coverage**	**L (s)**	**ME (m)**	**PS (# states)**	**Processing Time (ms)**	**Transitions**	**RP**	**SC**
frontend	5,392	91%	92	4.14	34	121.17	13	13,239	161,211
geo	5,743	89%	101	3.60	26	22.24	19	387	2,999
profile	5,556	87%	97	1.62	14	18.90	12	387	2,288
recommendation	5,665	91%	100	1.58	12	17.70	10	368	2,601
reservation	5,667	84%	100	2.34	17	18.60	12	400	2,332
rate	5,689	91%	100	1.74	14	18.28	11	380	2,002
search	5,312	91%	89	1.20	12	5.04	9	150	455
user	5,639	75%	100	1.60	14	24.88	10	404	4,449

TABLE 6: Evaluation statistics for protecting DeathStarBench's Hotel Reservation microservices (L=Lifting time, ME=Micro Execution time, PS=Policy Size, RP=Records Processed, SC=Number of system calls collected as relational flow records by the telemetry pipeline). Test suite generated 100k requests against the *frontend* microservice.

traffic sent to each service we use this data as test inputs for generating individual microservice effect graphs. Note that an input to the service may contain a combination of JSON, Protobufs, and URL-encoded request data. We use this data as test inputs for micro-executing each microservice. This produces an effect graph for each service that we provide to an MPM. The MPM then enforces all effect graphs across the entire microservice application. After enabling the MPM, we replay the final k segment of inputs, which were not known to micro execution. This allows us to empirically measure the rate of false positive alerts while protecting the hotel reservation application with an MPM. The results shown in the policy monitoring portion of Table 6 demonstrate that the MPM efficiently enforces effect graphs and that the reservation system's collective telemetry exercises all the microservices' effect graphs *without* incurring erroneous policy violations. Furthermore, the MPM processes records from all microservices. This shows the entire application is used throughout the test. Note that each individual record processed represents groups of system calls organized into operations and flows made on a specific resource (e.g., reading from and writing to a host's network port). This enables a substantial reduction (≈10x) in the number of runtime events that must be tracked, allowing the MPM to efficiently monitor the effects produced by the large volume of 100k requests made by the test suite.

6. Discussion

Effect Graph Applications. The MPM represents a single application of the effect graph model for security tasks. Effect graphs can also be transduced into mandatory access control and safety policies for kernel-enforced reference monitoring, such as policies enforced by AppArmor or seccomp (cf. Table 5), which would provide complete mediation and terminate the corresponding program as soon as a security policy violation occurs. Moreover, a separate analysis can automatically derive egress and ingress rules from an effect graph to inform a firewall running alongside the container to prevent issues like SSRF attacks.

Mimicry Attacks. Intrusion detection systems frequently fall victim to mimicry attacks [61], where an attacker crafts a system call sequence that conforms to a program's security policy but achieves malicious goals. Furthermore, these attacks can be automatically generated [62]. In this work, a successful mimicry attack causes malicious behavior while strictly conforming to a microservice' effect graph (e.g., by issuing API calls to permitted hosts). These mimicry attacks may pose a security concern if permitted communication paths can disclose sensitive information, or escalate privileges. Note the direction of communication is always specified in effect graphs and enforced by the MPM. This prevents adversaries from calling out to remote hosts, or trying to issue a `CONNECT` from microservices that only accept incoming connections. Maintaining accurate effect graphs can isolate individual services and limit the potential damage caused by these mimicry attacks.

Interpreted Languages. μPolicyCraft can model microservices implemented in interpreted languages (e.g., Python or JavaScript) by micro-executing an interpreter with the microservice container image as input. This enables the detection of effects incurred by the interpreter that may not be obvious by analyzing the microservice entrypoint on its own. In our experience, modeling the ABI for an interpreter like `cpython` requires significant effort. To simplify modeling interpreted programs, the interpreter could be modified directly to construct effect graphs during execution. Alternatively, the microservice could be rewritten such that an external library dynamically builds the effect graph by recording arguments passed to effectful methods as well as the location of these method calls within the program.

7. Conclusion

This paper presented μPolicyCraft, a system call specialization framework that synthesizes security policies for container images through symbolic micro execution, and detects policy violations over container telemetry. We presented an intermediate representation (IR) for the programs accepted by μPolicyCraft, outlined how μPolicyCraft produces effect graphs for binaries by micro executing this IR, and detailed our implementation and experiments. We evaluated μPolicyCraft on 25 challenges from the DARPA Cyber Grand Challenge (CGC) corpus, 5 real-world containers, and a complete microservice application from public benchmarks for which μPolicyCraft automatically produces effect graphs that can be enforced by a security monitor. We show that μPolicyCraft produces more precise policies compared to those produced by four existing system call specialization frameworks. Our results show that μPolicyCraft can model the effects of widely-deployed microservice architectures.

Acknowledgment

The research reported herein was supported in part by by U.S. ACC-APG / DARPA award W912CG-19-C-0003. Any opinions, recommendations, or conclusions expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for Public Release, Distribution Unlimited.

References

[1] S. Kongdee, "Introducing amazon cloudwatch container insights for amazon ecs," https://aws.amazon.com/blogs/mt/introducing-container-insights-for-amazon-ecs/, 2022, acc. 2022-08-03.

[2] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *Symposium on Recent Advances in Attacks and Defenses*, 2020.

[3] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, "Mining sandboxes for linux containers," in *IEEE International Conference on Software Testing, Verification and Validation*, 2017.

[4] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "sysfilter: Automated system call filtering for commodity software," in *Symposium on Recent Advances in Attacks and Defenses*, 2020.

[5] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen, "Automatic policy generation for inter-service access control of microservices," in *USENIX Security Symposium*, 2021.

[6] D. A. Wagner and D. Dean, "Intrusion detection via static analysis," in *IEEE Symposium on Security and Privacy*, 2001.

[7] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *International Symposium on Code Generation and Optmization*, 2004.

[8] MITRE, "CWE-918: Server-side request forgery (SSRF)," https://cwe.mitre.org/data/definitions/918.html, 2022, acc. 2022-08-17.

[9] P. Godefroid, "Micro execution," in *International Conference on Software Engineering*, 2014.

[10] C. Canella, M. Werner, D. Gruss, and M. Schwarz, "Automating seccomp filter generation for linux applications," in *ACM SIGSAC Cloud Computing Security Workshop*, 2021.

[11] S. Pailoor, X. Wang, H. Shacham, and I. Dillig, "Automated policy synthesis for system call sandboxing," in *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2020.

[12] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *USENIX Security Symposium*, 2020.

[13] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *International Conference on Computer-Aided Verification*, 2011.

[14] T. Taylor, F. Araujo, and X. Shu, "Towards an open format for scalable system telemetry," in *IEEE International Conference on Big Data*, 2020.

[15] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and Systems Security*, 2009.

[16] DARPA, "DARPA cyber grand challenge," https://github.com/CyberGrandChallenge/, 2021, acc. 2021-02-02.

[17] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[18] D. E. Denning, "An intrusion-detection model," in *IEEE Symposium on Security and Privacy*, 1986.

[19] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri, "Self-nonself discrimination in a computer," in *IEEE Symposium on Security and Privacy*, 1994.

[20] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *IEEE Symposium on Security and Privacy*, 1996.

[21] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, 1998.

[22] C. Warrender, S. Forrest, and B. A. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *IEEE Symposium on Security and Privacy*, 1999.

[23] A. Somayaji and S. Forrest, "Automated response using system-call delay," in *USENIX Security Symposium*, 2000.

[24] Y. Qin, S. Gonzalez, K. Angstadt, X. Wang, S. Forrest, R. Das, K. Leach, and W. Weimer, "Martini: Memory access traces to detect attacks," in *ACM SIGSAC Cloud Computing Security Workshop*, 2020.

[25] W. Lee and S. J. Stolfo, "Data mining approaches for intrusion detection," in *USENIX Security Symposium*, 1998.

[26] Guofei Jiang, Haifeng Chen, C. Ungureanu, and K. Yoshihira, "Multi-resolution abnormal trace detection using varied-length n-grams and automata," in *International Conference on Autonomic Computing*, 2005.

[27] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *IEEE Symposium on Security and Privacy*, 2001.

[28] G. Vigna and R. A. Kemmerer, "NetSTAT: A network-based intrusion detection approach," in *Annual Computer Security Applications Conference*, 1998.

[29] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous system call detection," *ACM Transactions on Information and Systems Security*, 2006.

[30] J. Byrnes, T. Hoang, N. N. Mehta, and Y. Cheng, "A modern implementation of system call sequence based host-based intrusion detection systems," in *IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications*, 2020.

[31] S. Das, Y. Liu, W. Zhang, and M. Chandramohan, "Semantics-based online malware detection: Towards efficient real-time protection against malware," *IEEE Transactions on Information Forensics and Security*, 2016.

[32] D. Gao, M. K. Reiter, and D. Song, "Gray-box extraction of execution graphs for anomaly detection," in *ACM Conference on Computer and Communications Security*, 2004.

[33] J. T. Giffin, S. Jha, and B. P. Miller, "Efficient context-sensitive intrusion detection." in *Network and Distributed System Security Symposium*, 2004.

[34] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller, "Formalizing sensitivity in static analysis for intrusion detection," in *IEEE Symposium on Security and Privacy*, 2004.

[35] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller, "Environment-sensitive intrusion detection," in *Symposium on Recent Advances in Intrusion Detection*, 2006.

[36] IETF, "Terminology, power, and inclusive language in internet-drafts and RFCs," https://datatracker.ietf.org/doc/draft-knodel-terminology/, 2023, acc. 2023-04-13.

[37] A. Bulekov, R. Jahanshahi, and M. Egele, "Saphire: Sandboxing PHP applications with tailored system call allowlists," in *USENIX Security Symposium*, 2021.

[38] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and Systems Security*, 2000.

[39] S. N. Chari and P.-C. Cheng, "Bluebox: A policy-driven, host-based intrusion detection system," *ACM Transactions on Information and Systems Security*, 2003.

[40] K. W. Hamlen, G. Morrisett, and F. B. Schneider, "Certified in-lined reference monitoring on .NET," in *ACM Workshop on Programming Languages and Analysis for Security*, 2006.

[41] I. Goldberg, D. Wagner, R. Thomas, E. A. Brewer *et al.*, "A secure environment for untrusted helper applications: Confining the wily hacker," in *USENIX Security Symposium*, 1996.

[42] K. Jain and R. Sekar, "User-level infrastructure for system call interposition: A platform for intrusion detection and confinement." in *Network and Distributed System Security Symposium*, 2000.

[43] N. Provos, "Improving host security with system call policies," in *USENIX Security Symposium*, 2003.

[44] AppArmor, "AppArmor," https://apparmor.net/, 2021, acc. 2021-02-02.
[45] SELinux Wiki, "SELinux wiki," https://selinuxproject.org/page/Main_Page, 2021, acc. 2021-02-02.
[46] J. Edge, "A seccomp overview," https://lwn.net/Articles/656307/, 2015, acc. 2022-08-03.
[47] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.
[48] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2007.
[49] "Mitre att&ck matrix for containers," https://attack.mitre.org/matrices/enterprise/containers/, 2021, acc. 2021-07-24.
[50] SysFlow, "Open telemetry and analytics pipeline," https://github.com/sysflow-telemetry, 2022, acc. 2022-08-03.
[51] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Software: Practice and Experience*, 1995.
[52] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy*, 2016.
[53] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *IEEE Symposium on Security and Privacy*, 2021.
[54] I. Gotovchits, "The BAP microexecution framework," https://opam.ocaml.org/packages/bap-primus/, 2022, acc. 2022-08-03.
[55] A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.
[56] M. Fleming, "A thorough introduction to eBPF," https://lwn.net/Articles/740157/, 2017, acc. 2022-08-03.
[57] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, 1991.
[58] Q-Success, "Usage statistics and market share for web servers," https://w3techs.com/technologies/overview/web_server, 2021, acc. 2021-11-14.
[59] OWASP, "OWASP top 10 - 2021," https://owasp.org/Top10/, 2022, acc. 2022-07-28.
[60] P. Burman, E. Chow, and D. Nolan, "A cross-validatory method for dependent data," *Biometrika*, 1994.
[61] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating mimicry attacks using static binary analysis," in *USENIX Security Symposium*, 2005.
[62] D. A. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *ACM Conference on Computer and Communications Security*, 2002.

Appendix A. Supplementary Material

A.1. Go ABI

In addition to supporting the ABI commonly used by C/C++ programs in POSIX environments, μPolicyCraft also supports modeling the Go ABI. Go is a popular statically typed programming language that is commonly used to implement individual services for micro-services. Similar to C ABIs, Go programs can be compiled into dynamically linked executables that use routines located in shared libraries located on a container image. Golang programs are often compiled into statically linked executables for production. In this work, we assume that Go programs are built as dynamically linked executables which is a compilation option supported by the production Go compiler.

Modeling Go programs requires accurately modeling the various components of the Go runtime. This includes the built-in Go types, such as strings, slices, and dictionaries. In addition, every Go executable maintains data structures for the garbage collected heap and checks that ensure memory reserved for local variables on the stack is not exhausted. Rather than modeling a completely accurate ABI, we develop policies for modeling specific fragments of the ABI that are necessary to produce a realistic model and use an optional Primus' feature to create a "random" ABI.

For example, Go functions often return a pair of values. The first may represent the return value for the function and the second an error struct that allows a caller to determine whether or not the function produced an error. To give the illusion that all Go functions return successfully, we intercept every external Golang function upon return and write `0` into the register that holds the error struct. This ensures that all external function calls appear successful, but the program may act on any data structures returned by the function. To ensure that data accesses or stores are performed successfully, we use Primus' ability to produce random values on invalid memory accesses to define a "probabilistic address space". This drastically reduces the human effort required to generate a model because, by default, all memory accesses and writes will succeed. In the case of an external Go function call, a random pointer will be returned as the function's return value, and any data read from that pointer will yield more random values. This ensures that micro-executing any program fragment always succeeds, even though it may lead to inaccurate models. An accurate model can be obtained by iteratively adapting portions of an ABI to be generated "randomly", by fixed policies, or by implementing functionality in either OCaml or Primus LISP. Our current prototype implementation includes portions of Go's standard library (i.e., `std`) and popular libraries used by the microservices found in our evaluation.

Moreover, Go features that support asynchronous programming, such as the `defer` statement, pose a challenge for our analysis. The `defer` statement allows a Go program to delay executing an expression until the end of a function. This is helpful for ensuring that resources such as files are always properly closed after use. However, excluding `defer` from our ABI produces incomplete models because every call to defer causes the Golang compiler to construct a continuation (i.e., some state that refers to where to jump before the function returns). Without a correct implementation of `defer`, the model will miss effects caused by deferred statements. μPolicyCraft can accurately model `defer` by capturing the destination address passed to the Golang runtime and scheduling a new Primus machine to start running from the destination address. Before the calling function returns, the current machine will stop, Primus will run the new machine on the continuation, and then return from the function. This allows μPolicyCraft to correctly model valid execution paths in programs that make use of asynchronous constructs like `defer`. Note that μPolicyCraft's implementation of built-in constructs alleviates the need for an analyst to be aware of them during modeling. From the analyst's perspective, micro execution proceeds as usual.

Appendix B. Meta-Review

B.1. Summary

The focus of this paper is system call specialization in microservice environments. The authors present the design, implementation, and evaluation of μPolicyCraft: a framework generating and enforcing security policies about system calls in microservices.

μPolicyCraft analyzes container images of microservices, extracts the binaries involved, and automatically generates, and subsequently enforces, a policy that encapsulates the allowed system calls of the corresponding process(es). More specifically, μPolicyCraft leverages the BAP framework to lift binary code to BAP IL, and then employs micro-execution to drive the code into executing benign system calls. During micro-execution, μPolicyCraft puts together a so-called effect graph, which is essentially a graph that describes benign state transitions, during the execution of the target program, with respect to system calls.

Once the effect graph has been extracted, μPolicyCraft proceeds with enforcing the mined policy, by executing the microservice atop a microservice-aware policy monitor (MPM), which leverages telemetry data to check if the system calls performed by the app follow allowed sequences.

μPolicyCraft is evaluated using 25 binaries from the DARPA CGC corpus, as well as five real-world apps, reporting numbers about the time needed for micro-executing the respective binaries, and size of the mined policy, etc. The paper concludes with various attack scenarios and how μPolicyCraft can detect the attack(s) involved.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) Valuable contribution to generating system call policies that take into consideration system call arguments, and a finite state machine using micro-execution, coupled with a well-defined formal construction and relatively low runtime overhead(s).
2) Effective solution that scales, solid systems approach to a relevant problem, and clear merit compared to prior work.

B.4. Noteworthy Concerns

1) The paper does not evaluate an interpreted application. Although μPolicyCraft can (successfully) execute the CPython interpreter, this is different than operating on an application implemented in an interpreted language.
2) The extent to which mimicry attacks are still possible is under-discussed.