ACE: Just-in-time Serverless Software Component Discovery Through Approximate Concrete Execution

Anthony Byrne
Boston University
abyrne19@bu.edu

Shripad Nadgowda
IBM T.J. Watson Research Center
nadgowda@us.ibm.com

Ayse K. Coskun
Boston University
acoskun@bu.edu

Abstract

While much of the software running on today's serverless platforms is written in easily-analyzed high-level interpreted languages, many performance-conscious users choose to deploy their applications as container-encapsulated compiled binaries on serverless container platforms such as AWS Fargate or Google Cloud Run. Modern CI/CD workflows make this deployment process nearly-instantaneous, leaving little time for in-depth manual application security reviews. This combination of opaque binaries and rapid deployment prevents cloud developers and platform operators from knowing if their applications contain outdated, vulnerable, or legally-compromised code. This paper proposes Approximate Concrete Execution (ACE), a just-in-time binary analysis technique that enables automatic software component discovery for serverless binaries. Through classification and search engine experiments with common cloud software packages, we find that ACE scans binaries 5.2x faster than a state-of-the-art binary analysis tool, minimizing the impact on deployment and cold-start latency while maintaining comparable recall.

CCS Concepts • **Software and its engineering** → *Empirical software validation*; Software maintenance tools; • **Computer systems organization** → **Cloud computing**.

Keywords serverless computing, software integrity, software component discovery

ACM Reference Format:
Anthony Byrne, Shripad Nadgowda, and Ayse K. Coskun. 2020. ACE: Just-in-time Serverless Software Component Discovery Through Approximate Concrete Execution. In *Workshop on Serverless Computing (WoSC '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3429880.3430098

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

WoSC '20, December 7–11, 2020, Delft, Netherlands

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8204-5/20/12…$15.00
https://doi.org/10.1145/3429880.3430098

1 Introduction

Today's ever-accelerating cloud software development cycles place a high cost on "reinventing the wheel," encouraging developers to use pre-built components (e.g., libraries, microservices, etc.) when they are available. While this practice saves time and money, any security vulnerabilities or reliability bugs within a pre-built component also become part of the resulting application. Failure to realize and properly manage these risks can have disastrous consequences: e.g., a vulnerability discovered in a popular web application framework led to the breach of 143 million Equifax customers' personal data [13]. Other than security concerns, external code may also be "undesirable" due to restrictive licensing that forces legally-binding provisions on any applications that include it (e.g., the GNU Affero General Public License [6]).

The risks inherent in pre-built components and opaque software packaging plague most forms of software, and serverless software is no exception. While most Function-as-a-Service (FaaS) platforms currently only host software written in interpreted languages, serverless container platforms such as Google Cloud Run, AWS Fargate, and Azure App Service offer the ability to automatically pull and run container images from a developer's registry or build server immediately after build completion. Because these serverless container platforms host software in the form of metadata-stripped[1] container-encapsulated binaries, they have a more difficult time screening out undesirable code than FaaS platforms, which have the luxury of possessing function source code that can be scanned using several well-established static analysis techniques [11]. This inability to discover undesired components of serverless software exposes both cloud developers and platforms to potential attacks, data breaches, and legal liability.

We envision a two-part solution to this software exploration problem: first, unlabeled binary blobs representing functions must be extracted from serverless applications and transformed into some kind of concise, uniquely-identifiable representation, which we refer to as a *function fingerprint*. Second, that fingerprint must be somehow checked for the existence of known undesirable software components in a process we call *software component discovery*.

[1]Executable Linkage Format (ELF) binaries contain optional metadata fields (e.g., compiler version, debugging symbols) that are routinely stripped during the build processes for many package formats [3]. Increasingly-popular single-binary "microcontainer" images also exacerbate this problem by obfuscating the OS resources included within a containerized application [16].

In this paper, we aim to protect serverless systems from buggy, vulnerable, or otherwise-unwanted code without significantly delaying deployment or increasing cold-start latency. To this end, we propose Approximate Concrete Execution (ACE), a just-in-time binary analysis technique that enables automatic detection of undesirable components in executable binaries found in serverless applications without requiring a trusted build system. With ACE, we contribute a novel method of creating function fingerprints just before serverless software is first-executed, in which we execute an intermediate representation of the code in an *approximate virtual machine* and use the resulting context as the fingerprint. As depicted in Fig. 1, ACE fingerprints can then be compared to a function blocklist using simple vector distance metrics or searched for in a k-nearest-neighbor fashion. In our evaluation, we find that ACE performs these tasks with comparable accuracy 5.2x faster than a state-of-the-art method.

We begin with a review of related binary analysis techniques in Section 2. After detailing ACE's methodology in Section 3, we present results from our initial experiments comparing ACE to the current state-of-the-art in Section 4. We then explore ACE's potential as a serverless binary fingerprinting tool and conclude with a discussion of our vision for ACE and potential future directions in Sections 5 and 6.

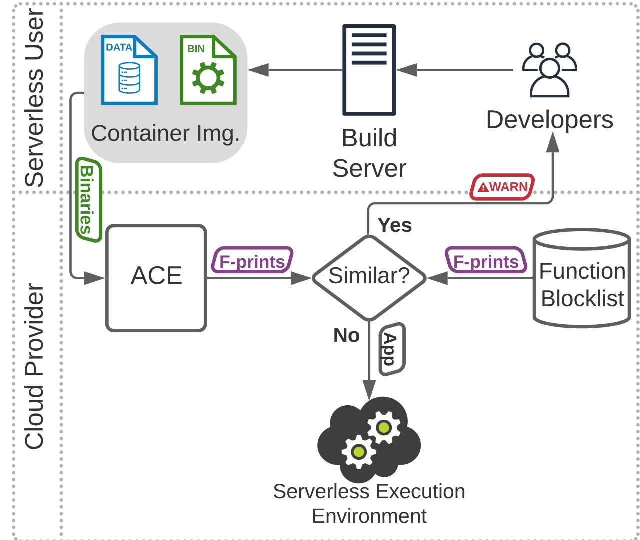

Figure 1. ACE can screen containerized serverless binaries just before execution. Binaries generating ACE fingerprints matching known "undesirable" code are flagged for review.

2 Background and Related Work

Like most modern software, cloud applications are often composed of both bespoke and off-the-shelf components that come together at various points in a software supply chain to produce a final build destined for deployment on cloud platforms. This build pipeline is frequently automated with continuous integration and deployment (CI/CD) tools like Jenkins and AWS CodeDeploy, which can automatically pull the latest versions of components from their source repositories, integrate them into the build environment, and deploy the final build — all within minutes or less. While instrumental in enabling the breakneck pace of development that users have come to expect, these highly-automated software supply chains can rapidly propagate the effects of human errors (e.g., accidental use of an outdated library) and have become attractive targets for bad actors looking to quietly insert backdoors, sabotage tools, and even cryptocurrency miners into otherwise-legitimate cloud software via third-party libraries or compromised compilers [2, 14, 17].

Discovering these undesirable components is a nontrivial problem: modern optimizing compilers produce variations in machine code that thwart simple binary pattern-matching methods, and as mentioned in Section 1, most of the metadata that could be used for component identification is frequently stripped out of the final binary to minimize storage and memory costs. How much information is left behind depends on how the binary is linked to its components. For dynamically-linked binaries, the stripping process does not remove the symbol table entries that reference the specific libraries in use. On the other hand, statically-linked binaries (like those produced by the increasingly-popular Golang compiler by default) do not require any symbols that reference external libraries to run, so most stripping tools remove all such metadata. Both processes blind cloud platforms to the contents of the binary serverless applications they host.

As outlined in Section 1, we propose a solution to this problem of restoring information lost during the serverless application build process in two phases: function fingerprinting and software component discovery. Below, we discuss approaches to both phases. We note that most of the prior binary analysis work we reference below does not specifically target cloud or serverless settings; nonetheless, we find the insights within this literature useful for developing our proposed solution.

2.1 Function Fingerprinting

Preprocessing. Individual functions can only be fingerprinted when in the form of discrete binary strings containing raw machine-code instructions. While extracting the executable binary file from its encapsulating container can be as easy as running `docker cp`, extracting properly-separated binary strings is often more difficult. This is especially true when the binary is stripped, as the symbols containing the start and end addresses of each function are removed [8]. Several approaches exist to restore this boundary information with high accuracy, from relatively-simple signature-based approaches (i.e., pattern matching) to advanced neural-network-assisted control flow graph analysis techniques [1, 10, 12, 19]. As in previous work, we use the ground-truth function boundary data found in DWARF debugging symbols embedded by the compiler in our experiments to ensure result integrity [4, 15].

Fingerprinting. Once the address boundaries of a function have been identified and its raw machine instructions extracted, most binary analysis approaches convert those instructions into a function fingerprint, or a concise uniquely-identifiable representation. In the popular disassembler IDA Pro's signature-based approach, this fingerprint is made of the first 32 bytes of the function (with some wildcards to account for compiler variations) combined with the CRC16 of a certain number of the following bytes [10]. Massarelli et al. propose "SAFE," a machine-learning-based approach inspired by natural language processing where each instruction "word" of a function is encoded and used to train a self-attentive neural network, which is in turn used to generate the function's fingerprint [15]. Egele et al. take an execution-based approach, in which the "side-effects" (i.e., register and memory values) caused by a function's execution on an x86 CPU become a function's fingerprint [5]. Pewny et al. take a similar symbolic-execution-based approach, where functions are disassembled, translated into an intermediate representation, and then converted into symbolic expressions [18]. Those symbolic expressions are executed with random concrete inputs, and the resulting input-output pairs become the function's fingerprint.

2.2 Software Component Discovery

After generating a fingerprint, a component discovery method must check the fingerprint for software components of interest. Methods that aim to discover *unknown* software security vulnerabilities, for example, accomplish this by mapping fingerprints to code samples containing generic classes of vulnerabilities, such as buffer overflows or use-after-frees [18]. Methods that focus on discovering *known* undesirable components (e.g., "cryptojacking" libraries or restrictively-licensed code) accomplish this by mapping fingerprints to a corpus of libraries and function names [10, 12, 20]. In this paper, we focus on the latter application of discovering known components, which we term *function fingerprint classification.* Below, we compare some recent approaches in function fingerprint classification.

Signature-based approaches like those used by IDA Pro tend to have high precision but low recall, as they can only discover functions that match a byte-signature in their databases [1]. Since this search is exhaustive and repeated for each function in a binary, and since each function may require multiple signatures due to compiler variations, these databases are typically limited to the most common library functions [10].

Execution-based approaches, on the other hand, tend to have high recall but low precision. Pewny et al. and Egele et al. all evaluate their work as "binary search engines," measuring accuracy in terms of how high a query function fingerprint's true label ranks among the top k most-likely labels returned by the search engine (where k varies between 10 and 200) in a k-nearest-neighbor-style fashion [5, 18]. Using this metric, Egele et al.'s and Pewny et al.'s methods achieve 77% and 56% accuracy, respectively.

Learning-based approaches vary greatly in their performance. Some approaches achieve very high accuracy at the cost of overhead high enough to disqualify use in a "just-in-time" application like serverless software component discovery [9]. Xu et al.'s method "Gemini," for example, proposes a highly-accurate neural-network-assisted fingerprinting approach, but it can take up to 10 GPU-hours to train the network on 100,000 function samples [20]. In Section 4, we evaluate the current state-of-the-art learning-based approach, SAFE, and find that its high overhead outweighs the benefits of its high accuracy for serverless applications [15].

3 ACE: Approximate Concrete Execution

With ACE, we attempt to combine some of the lessons learned from the successes of recent works to create a low-overhead serverless binary function fingerprinting and classification method that achieves acceptable accuracy and high tolerance to variations across compiler versions and optimization levels. In practice, the fingerprints produced by ACE will be labeled, cataloged, and compared against fingerprints on a "function blocklist" pre-execution, as depicted in Fig. 1. A match between an unlabeled fingerprint and a blocklist entry would indicate that potentially undesirable code is present within the binary in question.

With these goals in mind, we apply the key insight developed by Pewny et al.'s and Egele et al.'s execution-based approaches: given known and consistent inputs, a function $F()$ compiled by compiler C for architecture A will have a demonstrably-similar effect on its environment (i.e., the CPU context) when executed as that same function $F()$ compiled by compiler C' for architecture A' [5, 18]. We apply this insight by executing functions in an approximate virtual machine and using the resulting context as the function's fingerprint.

How one defines "demonstrably-similar," of course, is highly dependent on one's application. For our target application of just-in-time serverless binary scans, we envision different cloud users having different levels of security-consciousness: e.g., developers of an online banking application may be especially tolerant of the occasional false positive in exchange for catching more truly undesirable code, while developers of a weather application may only wish to be alerted when the presence of unwanted code is all but guaranteed. Therefore, for the first experiment we describe in this paper, we define "demonstrably-similar" to mean "exact match" (i.e., the definition least likely to produce false positives). In our second experiment, however, we expand this definition to include fingerprints similar enough that a statistical search method (e.g., k-nearest-neighbor) returns $F()$'s label as one of the top-k most-likely classifications of $F()$'s fingerprint (i.e., the definition least likely to produce false negatives). In our future work, we plan to explore the use of other machine learning methods (e.g., logistic regression) in our software component discovery process.

3.1 Binary Preparation

As explained in Section 2.1, we begin by extracting ELF binaries from their encapsulating containers and breaking them down into their individual functions using the boundary address data embedded within the DWARF debugging symbols by GCC [4]. Then, for each raw binary function, we follow the process shown in Fig. 2, beginning with the disassembly of the functions into their native assembly code. In our implementation, we utilize the Capstone disassembler due to its speed and multi-platform support.

3.2 Intermediate Representation

In order to mitigate the idiosyncratic differences in binaries compiled by different compilers, compiler versions, optimization levels, etc., the now-disassembled function must be translated into some common architecture- and compiler-agnostic representation: i.e., an *intermediate representation* (IR). In our implementation of ACE, we choose Zynamics' Reverse Engineering Intermediate Language (REIL) as our IR due to its small RISC-like instruction set (17 instructions vs. 1,503 in x86), simple structure (each instruction has exactly 3 operands and returns 1 value), and availability of translator source code for x86, ARM, and PowerPC (to allow for the possibility of cross-platform fingerprinting in the future) [21].

3.3 Approximate Virtual Machine

Although REIL serves as a platform-agnostic common language, two code-identical, byte-different functions (e.g., identical code compiled at different optimization levels) will typically not produce the same REIL code when translated, making it unsuitable as a function fingerprint. Instead, we execute the REIL representation of a function inside an *approximate virtual machine* (aVM) and collect the final state of the aVM as the function's fingerprint. Applying the insight discussed above, this execution-based process should produce a fingerprint that satisfies the "demonstrable-similarity" property.

We define ACE's aVM as a simple instruction-level emulator for the REIL instruction set. However, unlike most emulators, the aVM's goal is not to accurately emulate the theoretical REIL CPU, which Zynamics defines as a "CPU with unlimited memory and an unlimited number of registers" that executes instructions sequentially without side-effects [21]. Instead, the novel aVM is designed to produce consistent, demonstrably-similar fingerprints for code that is similar, invariant of noise like compiler optimizations or toolchain version differences.

In order to mitigate certain compiler optimizations like instruction re-ordering, the aVM first sorts REIL instructions lexicographically.[2] The aVM is then initialized to some known state, e.g., with all registers and memory set to zero. Finally, the aVM sequentially executes each instruction "approximately," i.e., with a loose adherence to its specified operation, aggressively optimizing for speed and simplicity over exact emulation. For example, traditionally-slow control flow instructions (e.g., conditional-jump or `JCC`) are treated as null instructions or `NOP`s.

This relaxed accuracy constraint sets ACE apart from the other execution-based methods discussed in Section 2.1, which attempt to accurately model the function's native architecture when generating fingerprints, often at the cost of additional overhead. It also greatly simplifies the aVM, with our current implementation containing only 375 lines of Python code.

After executing all instructions in a REIL code function, the aVM returns its final state as the function's fingerprint. In our current implementation, only the values of the first 32 registers are returned (instructions involving registers beyond the 32nd are nullified), making every fingerprint a 32-element vector of integers. We are currently exploring ways to incorporate the aVM's "unlimited memory" into ACE's fingerprints.

We repeat this process for every function in a binary. Since each function can be fingerprinted independently of any other function, this process is embarrassingly-parallelizable, and we take advantage of that property through multithreading in our implementation.

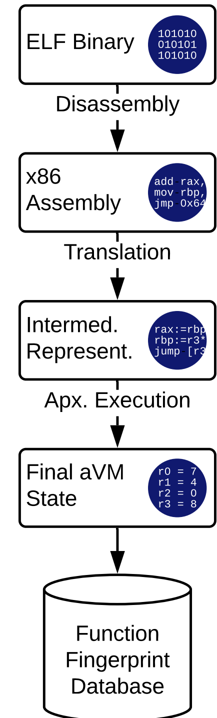

Figure 2. ACE fingerprinting process overview.

4 Evaluation

Our goal for evaluation in this paper is to assess the feasibility of a just-in-time serverless software component screening tool using ACE as its function fingerprinting method. To this end, we perform two experiments: the first assesses ACE's basic ability to *specifically* identify a piece of code we inject into 228 applications commonly found in cloud systems (similarly to how one might search for a buggy function present in a popular cloud library), and the second assesses ACE's ability to discover functions more *sensitively* (i.e., with a progressively-lower confidence threshold) in an attempt to achieve better recall in the presence of several different optimization levels and compiler versions. In both experiments, we compare ACE to a state-of-the-art approach, SAFE [15].

4.1 Datasets

Our evaluation dataset for our code-injection experiment consists of 228 64-bit x86 ELF binaries drawn from 4 packages

[2]The choice of sorting key is arbitrary and inconsequential provided it is consistent across all fingerprints in a dataset.

found in most Linux-based systems (coreutils, util-linux, diffutils, and inetutils) and the MediaService application from the *DeathStarBench* cloud benchmarking suite [7]. We choose these applications to achieve diversity in terms of binary size and functionality. We modify all Makefiles to disable the stripping of binaries, as we require a ground-truth source of function names and address bounds. We compile all applications using the standard GCC toolchain in an Ubuntu 18.04 build environment, and the resulting binaries contain 18,567 functions in total, including external library functions.

For our function search experiment, we use a modified version of the "AMD64PostgreSQL" dataset published by Massarelli et al [15]. This modified dataset contains 344,396 functions extracted from 28 binary versions of PostgreSQL, a popular cloud database, each compiled by a different version of GCC (versions 3.4, 4.7, 4.8, 4.9, 5.4, 6, and 7) at four different optimization levels (O0, O1, O2, O3).

4.2 Code-injection Experiment

In order to function as an effective serverless software component screening tool, ACE should generate a consistent fingerprint for a given piece of undesirable code regardless of compiler variations. To assess this ability, we inject C and C++ functions with known ACE fingerprints[3], shown in Fig. 3, into the main source code file of every application in our dataset. We compile these "injected" applications alongside their "clean" counterparts using the process outlined in Section 4.1, creating the set of 456 binaries (containing 37,134 functions) that we use for this experiment. Note that due to compiler variations, there exist 217 unique versions of the binary string representation of dummy_math()'s instructions, i.e., most instances of dummy_math() are *not* byte-identical to other instances of the function.

We begin our evaluation by generating ACE and SAFE fingerprints for every function in the set in parallel on an 8-vCPU virtual machine, using the process defined in Section 3 for ACE and the code published by Massarelli et al. for SAFE. Following typical binary classification protocols, we classify each function as dummy_math or not_dummy_math based on whether or not the ACE/SAFE fingerprint produced by the function is an exact match to the known fingerprint for dummy_math(). We count the number of Type I and II errors that occur and calculate the resulting precision, recall, and F1 score for use as performance metrics. As shown by the results in Fig. 4, ACE achieves near-perfect accuracy, generating the correct fingerprint in all but two cases.

4.3 Function Search Experiment

In order to evaluate ACE's recall in the presence of serverless applications compiled by different toolchains, and to

[3]Despite backward-compatibility, C compilers interpret the semantics of the function shown in Fig. 3 significantly differently than C++ compilers, leading us to classify each language variant separately in this experiment. We leave language-agnosticism to future work.

```
1  int dummy_math() {
2    volatile int num, nabs, result;
3    volatile div_t ndiv;
4    volatile int array[16];
5    unsigned short int i;
6    for (i=0; i < 16; i++) {
7      num = rand() / (RAND_MAX / 360);
8      array[i] = (i > 0) ?
9        (array[i-1] + num) : num;
10   }
11   if (num > 100) nabs = abs(num + 1);
12   else nabs = abs(num - 1);
13   ndiv = div(nabs, 7);
14   result = 2 * (num * ndiv.quot) - nabs;
15   return result;
16 }
```

Figure 3. C/C++ code of the dummy function used in the code-injection experiment (Sec. 4.2). We use the **volatile** keyword to dissuade the compiler from "optimizing away" the function, as we do not add calls to it.

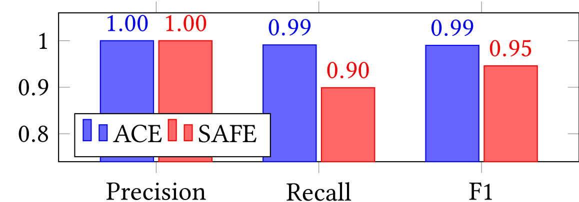

Figure 4. Code-injection experiment (Sec. 4.2) accuracy comparison. Note that only exact-fingerprint-matches are considered true positives.

more-directly compare ACE against a current state-of-the-art method, we recreate the "function search" experiment described by Massarelli et al. using the modified dataset described in Section 4.1. We begin by generating ACE fingerprints for each PostgreSQL function similarly to the previous experiment. Instead of generating SAFE fingerprints ourselves, however, we use the fingerprints provided in the AMD64PostgreSQL dataset.

We then generate an 80,000-function query set composed of 10,000 functions randomly sampled from each compiler-version/optimization-level combination. For each function in the query set, we find the top k ACE/SAFE fingerprints nearest (in terms of Euclidean distance) to the query function's ACE/SAFE fingerprint, where $k \in \{10,20,30,40,50,100,200\}$. We consider functions with the same name and origin filename as the query function to be true positives when returned by the query or false negatives when not returned. Using recall as our performance metric, we find that while SAFE delivers recall scores 3-10% higher than ACE for $k < 100$, ACE outperforms SAFE by similar margins for $k \geq 100$, reaching a maximum recall of 0.67. Furthermore, we find ACE has a significantly lower end-to-end runtime than SAFE, as shown in Table 1.

Table 1. Function Search Experiment Overhead Comparison

Phase	Runtime (min) ACE	SAFE
Function Fingerprinting	3.03	5.87*
Software Component Discovery	1.10	15.65
Total	**4.13**	**21.52**

*Pre-fingerprinted dataset used for SAFE, so fingerprinting runtime shown is interpolated from runtime reported by [15].

5 Discussion

The evaluation detailed in Section 4 leads to a few key insights. First, the results of the code injection experiment show that the function fingerprints ACE creates are in most cases identical for code-identical functions, even when the machine instructions for those functions differ. As discussed in Section 3, this fingerprint similarity property is key to a successful execution-based function fingerprinting tool, meaning ACE shows serious potential as a serverless software component discovery method.

Second, we highlight the sizable (5.2x) difference in speed between ACE and SAFE described in Table 1. We also note that unlike SAFE and other learning-based methods, ACE requires no expensive pre-training, further reducing overhead and the need for frequent model updates. These qualities make ACE well-suited for use as a just-in-time serverless software component discovery tool, as its use would cause only minimal impacts to cold start latency or deployment delay compared to other methods.

Third, we find that ACE's recall follows a logarithmic growth trend (with respect to increasing values of k) largely similar to that of the state-of-the-art method in the function search experiment, described in Section 4.3. This indicates that ACE is responsive to sensitivity threshold "tuning," which is important for suiting different users' levels of security-consciousness, as mentioned in Section 3.

Lastly, we note that there is room for accuracy improvements to ACE, especially when it comes to reducing "fingerprint collisions." These occur when ACE generates identical fingerprints for two or more similar-but-not-identical functions, leading to lower recall in function search scenarios as false-positive fingerprints "crowd out" true-positive ones. We note that there are many tuning variables under our control, such as initial aVM state, register file size, and memory data usage, that could be customized for better accuracy.

6 Conclusion

In this paper, we propose Approximate Concrete Execution (ACE), a method of generating binary function fingerprints to facilitate the discovery of undesirable software components in serverless container images. Our experiments show that ACE has significant promise as a very-low-overhead fingerprinting method that could be used as a just-in-time component discovery tool while only minimally impacting performance.

Acknowledgments

This work is partially funded by IBM Research.

References

[1] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-Agnostic Function Detection in Binaries. In *2017 IEEE Euro. Symposium on Sec. and Priv.* IEEE, 177–189. https://doi.org/10.1109/EuroSP.2017.11
[2] M. S. Day and C. A. Jennings. 2018. *22 Thompson's Hack.* 263–272. https://ieeexplore.ieee.org/abstract/document/8555305
[3] Debian Project. 2019. Binaries. https://perma.cc/22DT-ZDZV
[4] DWARF Committee. 2017. The DWARF Standard. http://dwarfstd.org
[5] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In *Proc. of the 23rd USENIX Sec. Symposium.* 303–307.
[6] Free Software Foundation. 2020. Freqently Asked Questions about the GNU Licenses. https://www.gnu.org/licenses/gpl-faq.html
[7] Yu Gan et al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proc. of the 24th Intl. Conf. on Arch. Support for Prog. Langs. and Operating Sys.* ACM Press, 3–18. https://doi.org/10.1145/3297858.3304013
[8] Laune C. Harris and Barton P. Miller. 2005. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News* 33, 5 (Dec. 2005), 63. https://doi.org/10.1145/1127577.1127590
[9] Jingxuan He et al. 2018. Debin: Predicting Debug Information in Stripped Binaries. In *Proc. of the 2018 ACM SIGSAC Conf. on Comput. and Comms. Sec. (CCS '18).* ACM, New York, NY, USA, 1667–1680. https://doi.org/10.1145/3243734.3243866
[10] Hex-Rays SA. 2020. IDA F.L.I.R.T. https://perma.cc/JVC8-E3A9
[11] Thomas Hofer. 2010. Evaluating Static Source Code Analysis Tools. (2010). http://infoscience.epfl.ch/record/153107
[12] Emily R. Jacobson, Nathan Rosenblum, and Barton P. Miller. 2011. Labeling library functions in stripped binaries. In *Proc. of the 10th ACM SIGPLAN-SIGSOFT Workshop on Prog. Analysis for Software Tools.* ACM Press, 1. https://doi.org/10.1145/2024569.2024571
[13] Jeff Luszcz. 2018. Apache Struts 2: How Technical and Development Gaps Caused the Equifax Breach. *Network Security* 2018, 1 (Jan. 2018), 5–8. https://doi.org/10.1016/S1353-4858(18)30005-9
[14] Tim Mackey. 2018. Building open source security into agile application builds. *Network Security* 2018, 4 (April 2018), 5–8. https://doi.org/10.1016/S1353-4858(18)30032-1
[15] Luca Massarelli et al. 2019. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In *16th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2019).* Springer, Cham, 309–329. https://doi.org/10.1007/978-3-030-22038-9
[16] Shripad Nadgowda, Sahil Suneja, and Canturk Isci. 2018. RECap: Run-Escape Capsule for On-demand Managed Service Delivery in the Cloud. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18).* USENIX Association, Boston, MA, USA.
[17] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Lecture Notes in Computer Science.* Vol. 12223 LNCS. 23–43. https://doi.org/10.1007/978-3-030-52683-2_2
[18] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-Architecture Bug Search in Binary Executables. In *2015 IEEE Symposium on Security and Privacy.* IEEE, 709–724. https://doi.org/10.1109/SP.2015.49
[19] E. C. R. Shin, D. Song, and R. Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Security Symposium.* USENIX Association, Washington, D.C., 611–626.
[20] Xiaojun Xu et al. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proc. of the 2017 ACM SIGSAC Conf. on Comput. and Comms. Sec.* ACM Press, 363–376. https://doi.org/10.1145/3133956.3134018
[21] Zynamics. 2011. REIL. https://perma.cc/LW4F-ZSYW