Secure Containers in High-Performance Computing Infrastructure

Overview

The security and privacy of high-performance computing (HPC) infrastructures are critically important as HPC infrastructures often process sensitive data, perform important scientific computations, and are relied upon by many organizations and individuals. Containers, as lightweight and isolated environments for running applications, e.g., Docker, and Singularity, are becoming increasingly popular in HPC infrastructures, while their security problems are paid less attention. Particularly, the containers in HPC infrastructure mainly face two security problems. **(i)** The container images are insecure. E.g., a recent study on 44 neuroscience container images of both Docker and Singularity shows that there are ***460 vulnerabilities per image***. **(ii)** The weak isolation may lead to vulnerabilities caused by running multiple containers on the same OS kernel. We observed ***11 such vulnerabilities since 2017***.

In this proposal, we aim to design secure containers for HPC infrastructures. **(i)** To address the insecure image problem, existing container image vulnerability scanners, e.g., Clair, Trivy, and Grype, face a *low coverage* challenge as they mainly conduct software version-based look-ups in public vulnerability databases. Therefore, in *Thrust 1*, we propose to design an efficient image vulnerability scanner with various innovative and feasible techniques, e.g., language-agnostic code representation with IR-reoptimization and natural language processing (Task 1-1), code similarity detection with graph neural network and triplet-loss network (Task 1-2), and scalable online search with locality-sensitive hashing (Task 1-3). **(ii)** Existing sandboxed runtime-based container solutions can address container isolation vulnerabilities by introducing a translation layer between the container and host kernel while facing low performance issues for HPC workloads. On the other hand, existing HPC container solutions (e.g., Docker, Singularity, Charliecloud, and Shifter) share the kernel with the host and are demonstrated to be vulnerable. Therefore, in *Thrust 2*, we propose to develop a secure and high-performance container runtime by using a lightweight virtual machine hypervisor (Task 2-1) with various customized optimizations for security and performance towards HPC workloads (Task 2-2), and dynamic image debloating to further remove attack surfaces (Task 2-3).

Intellectual Merit

Our project advances the security of containers in HPC infrastructure by offering a secure container platform for efficiently and securely running containers in HPC infrastructures. Our novel contributions are two-fold, i.e., an efficient image vulnerability scanner, and a secure and high-performance container runtime. *We envision this proposed work can significantly reduce the attack surfaces of containers in HPC infrastructures. Not limited, we aim to broaden the real impacts of this project by integrating our techniques and tools with existing HPC stakeholders to facilitate the security and privacy of HPC infrastructures.*

Broader Impacts

This research can have the following broader impacts. **(i)** The research will significantly advance the security of containers and thus the security of HPC infrastructures. The PIs plan to conduct system integration with HPC stakeholders; **(ii)** The PIs will integrate the proposed research into their curriculum development at both graduate and undergraduate levels. This proposed project will foster new research and educational opportunities at both the University of North Texas (UNT), a **Minority Serving Institution (MSI)** and a **Hispanic Serving Institution (HSI)**; **(iii)** The PIs will conduct outreach and educational activities in the K-12 community and promote the participation of students from underrepresented groups in Cybersecurity and HPC. **(iv)** In addition to disseminating the research results through publications, the PIs will also create a webpage and a publicly accessible GitHub repository to share the research results.

1 Introduction

High-performance computing (HPC) infrastructure aims to provide fast, reliable, and efficient computational resources to support cutting-edge scientific research and engineering applications, including weather forecasting [1], climate modeling [2], molecular simulations [3], and cryptography [4]. To avoid the tedious human efforts and unexpected errors of configuring the execution environment for running jobs on HPC infrastructure, containers, as lightweight and isolated environments, e.g., Docker [5], and Singularity [6], have been deployed in an increasing number of HPC servers, e.g., Summit [7], Sierra [8], and Frontera [9].

The security and privacy of HPC infrastructures are critically important for three main reasons [10, 11]. **(i)** *Confidentiality*. HPC systems are often used to process sensitive data, including classified information, financial data, and personal information. The breach of such data would cause serious societal crises. E.g., in 2020, the University of Utah reported a data breach in which an attacker accessed a server with sensitive personal information [12]. **(ii)** *Integrity*. HPC systems are performing important calculations, e.g., simulating nuclear reactions or designing new drugs. Intentionally injected errors or tampering can have serious negative consequences, e.g., blocking scientific innovation or leading scientists in the wrong direction. E.g., in 2018, a cybercrime gang attacked a Russian biochemical research institute and altered data on the effects of drug treatments [13]. **(iii)** *Reliability*. HPC systems are critical infrastructures relied upon by many organizations and individuals. The downtime can have significant impacts. E.g., in 2020, about a dozen HPC servers were shut down across Europe due to security incidents [14], leaving researchers unable to work.

The need of securing containers in HPC. Though containers have become essential components in HPC infrastructure, their security problems are paid less attention to, which can potentially lead to serious cyber attacks. Particularly, the containers in HPC infrastructure mainly face two security problems as listed below.

Problem 1: Insecure images. A container is usually built from an image configured by a domain scientist or engineer, who can freely choose a base image and install whatever software he or she needs. As container images often include outdated or unnecessary software packages, they can introduce a large number of vulnerabilities. For example, a recent study on 44 neuroscience container images of both Docker [5] and Singularity [6] shows that there are ***460 vulnerabilities per image***, allowing remote attackers to execute arbitrary code, steal user credentials, and steal sensitive data [15].

Problem 2: Container isolation vulnerability. Containers are isolated from the host system and other containers, relying on Linux kernel features. However, the isolation may not be perfect as the shared kernel inevitably exposes attacking surfaces. There is a risk that containers can interfere with the host system or other containers, potentially compromising security [27]. E.g., containers can escape isolation and escalate privileges. Table 1 surveys ***11 container escape vulnerabilities***, mostly caused by vulnerabilities in the shared kernel. Also, when running multiple containers (presumably belonging to different tenants) on the same kernel, the shared kernel enables adversary containers to mount various attacks, e.g., data breach [28, 29], resource exhaustion [30, 31], and denial-of-service [32, 33].

Table 1: Container escape vulnerabilities since 2017.

Vulnerability (CVE)	Location	Description
2022-0185 [16]	Kernel	Buffer Overflow
2022-0492 [17]	Kernel	Improper access restriction
2022-23648 [18]	containerd	Improper image volume handle
2022-0847 [19]	Kernel	Read-only files overwritten
2021-30465 [20]	runC	Race condition
2021-22555 [21]	Kernel	Heap out-of-bounds write
2021-31440 [22]	Kernel	Incorrect bound check
2020-8835 [23]	Kernel	Improper ebpf program verify
2019-5736 [24]	runC	Improper file descriptor handle
2017-5123 [25]	Kernel	Insufficient data validation
2017-1000112 [26]	Kernel	Memory corruption

Thus, we urgently need security mechanisms that can not only detect vulnerabilities in container images but also ensure the secure execution of containers in HPC infrastructures.

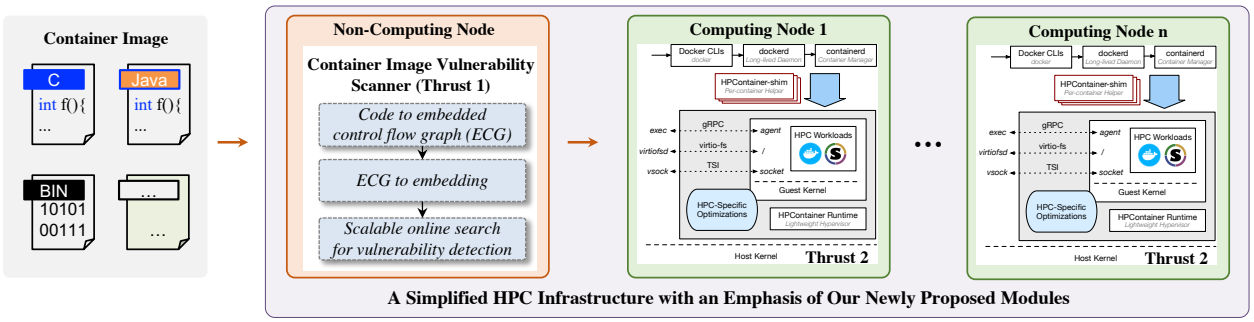

Figure 1: Overview of the proposed work in a simplified HPC infrastructure. Once a container image is uploaded, our container image vulnerability scanner (**Thrust 1**) can quickly detect the vulnerabilities inside it, which can be installed in a non-computing node and launched during the waiting period of that task. Later, when the container image executes, it will run inside our newly proposed HPContainer runtime (**Thrust 2**) on each computing node.

Unfortunately, existing container secure mechanisms face two challenges for HPC infrastructure.

Challenge 1: Low coverage of detecting vulnerabilities in container images. To address the insecure image problem (**Problem 1**), existing container image vulnerability scanners, e.g., Clair [34], Trivy [35], and Grype [36], mainly conduct software version-based look-ups in public vulnerability databases. Due to that, they face a major challenge of low coverage as they can not detect *unknown software*, e.g., internally developed. Since the scientists frequently develop and run internally developed software in HPC infrastructure, that leaves a huge gap between existing image scanners and the actual requirements in HPC infrastructure.

Challenge 2: Secure and high-performance container runtime. Existing *sandboxed runtime*-based container solutions, mostly used by cloud vendors, can address container isolation vulnerabilities (**Problem 2**). They introduce a translation layer between the container and the host kernel. Though they are more secure, they cannot achieve satisfactory performance for HPC workloads [37, 38, 39]. On the other hand, existing HPC container solutions (e.g., Docker [5], Singularity [40], Charliecloud [41], and Shifter [42]) largely utilize *bare-metal runtime*, where the containers share the kernel with the host. Though they can achieve high performance, they have been demonstrated to be vulnerable to attacks as discussed in Problem 2.

Proposed research. Figure 1 overviews our newly proposed work in a simplified HPC infrastructure. In particular, it addresses the above challenges through the following two research thrusts.

Thrust 1 designs an efficient image vulnerability scanner to improve the coverage (Section 4), which can be deployed on a non-computing node and leverage the waiting period of a task to scan the image as shown in Figure 1. We propose to seek a code similarity approach as it can achieve high coverage. However, existing methods can not be directly applied as *they are usually limited to one code type*, e.g., C/C++. Differently, in a container image, there exists *a wide mix of various types of code*, e.g., C/C++, Java, Python, and binary code compiled from various languages. In particular, we propose three research tasks as discussed below.

- *Task 1-1* proposes a language-agnostic code representation to cover various types of code with the help of LLVM-IR, IR re-optimization, and natural language processing (NLP) techniques (Section 4.1).
- *Task 1-2* proposes to use graph triplet-loss network, a combination of graph neural network and triplet-loss network, to efficiently generate code embedding for code similarity detection (Section 4.2).
- To further improve the scalability, *Task 1-3* designs an efficient locality-sensitive hashing-based online search method together with conventional image scanners (Section 4.3).

Thrust 2 develops a secure and high-performance container runtime (Section 5), which can be deployed on each computing node as shown in Figure 1. Existing approaches either choose bare-metal methods that can achieve near-native performance, but *contain a series of vulnerabilities*; or utilize sandboxed runtimes that are more secure, but largely *sacrifice the performance*, especially for HPC workloads. In particular, we propose three research tasks as discussed below.

- *Task 2-1* proposes to use a lightweight virtual machine hypervisor as the container runtime with various optimizations of virtual resource management and system latency reduction (Section 5.1).
- To improve the security and performance, *Task 2-2* proposes to customize the runtime based on HPC requirements from four levels, i.e., hypervisor feature, file system, network, and GPU (Section 5.2).
- To further reduce the attack surfaces, *Task 2-3* proposes to design a dynamical image debloating method that can remove unnecessary files, software, and packages (Section 5.3).

Intellectual merit. Our project advances the security of containers in HPC infrastructure by offering a secure container platform for efficiently and securely running containers in HPC infrastructures. Our novel contributions are two-fold. **(i)** We propose to design an efficient image vulnerability scanner with various innovative and feasible techniques, e.g., language-agnostic code representation with IR-reoptimization and NLP, code similarity detection with graph triplet-loss network, and scalable online search with locality-sensitive hashing. **(ii)** We propose to develop a secure and high-performance container runtime by using a lightweight virtual machine hypervisor with various customized optimizations for security and performance towards HPC workloads. *We envision this project can significantly reduce the attack surfaces of containers in HPC infrastructures. More importantly, we aim to broaden the real impacts by integrating our techniques and tools with existing HPC stakeholders to facilitate the security and privacy of HPC infrastructure.*

Broader impacts. (i) The research will significantly advance the security of containers and thus the ecosystem of high-performance computing (HPC) infrastructures. The PIs plan to conduct the system integration together with HPC stakeholders, e.g., University of North Texas (UNT) HPC Services, Texas Advanced Computing Center (TACC), and National Oceanic and Atmospheric Administration (NOAA) (more details in Section 6). **(ii)** The PIs will integrate the proposed research into their curriculum development at both graduate and undergraduate levels. These efforts will meet the growing national need for professionals in Cybersecurity and HPC. This proposed project will foster new research and educational opportunities at both UNT, a **Minority Serving Institution (MSI)** and a **Hispanic Serving Institution (HSI)**, and the University of Delaware (UD), where Delaware is an EPSCoR state in great need of education and research activities. **(iii)** The PIs will conduct outreach and educational activities in the K-12 community and promote the participation of students from underrepresented groups in Cybersecurity and HPC.

Team qualifications. Our team is uniquely positioned to complete the proposed work, which requires a comprehensive understanding of container, software security, system security, and high-performance computing (HPC). PI Ji (UNT) specializes in both software security and HPC, especially for software vulnerability detection. PI Gao (UD) specializes in system security and cloud computing, especially container security. The two PIs have published various related papers in prestigious security and HPC venues, such as IEEE S&P [43, 44], Usenix Security [45, 46, 47], CCS [30, 48], NDSS [49], SC [50], and HPDC [51, 52]. Senior personnel Herrington (UNT) is an advanced computing services consultant at UNT. He is the main point of contact for UNT computing service and the Texas Advanced Computing Center (TACC) from the UNT side. The two PIs and senior personnel have been actively collaborating on container security in HPC infrastructure. The papers are still work-in-progress.

2 Relevance to CICI

The proposed work advances the security and privacy of scientific computation by securing the frequently used ***containers in cyberinfrastructure***. The resulting methodologies, systems, and open-source contributions of this work will support scientific computation practitioners and researchers with techniques and tools to protect their software, data, and cyberinfrastructure. We will actively collaborate with scientific computation practitioners, researchers, and stakeholders to encourage the adoption of our techniques and tools into the scientific workflow, and help to create a holistic, integrated security environment spanning the entire scientific cyberinfrastructure ecosystem. To this end, we believe this project fits perfectly well with the Usable and Collaborative Security for Science (UCSS) within the CICI program.

3 Background

Containers in HPC. Containers have been widely used in HPC infrastructures. Containers such as Docker [5] wrap the application code, its dependencies, and libraries into one image that can be easily deployed and run on any compatible system. Also, as a lightweight operating system-level virtualization technique, containers typically share the same host kernel and can achieve near-native performance [53, 54, 55] by fully relying on multiple building blocks in the Linux kernel for resource isolation and control. While Docker is one of the most popular container technologies, to meet the special performance and security needs of running HPC workloads, several other container technologies have been developed, including Singularity [40], and Shifter [42]. Singularity [40] is an open-source project developed by Lawrence Berkeley National Laboratory. It has been widely adopted in the HPC community as it can support most parallel processing technologies, and easily integrate with existing HPC job schedulers and resource managers. Shifter [42], sharing similar features to Singularity, is developed by the National Energy Research Scientific Computing Center (NERSC), a division of the U.S. Department of Energy Office of Science. From the developers' (scientists or engineers) perspective, they usually use Docker on their local machines during the development and testing phase. When the Docker image is ready to run on the HPC servers, they convert it to other containers, e.g., Singularity or Shifter, upload it to the server, and run it.

Container image vulnerability and scanner. A container image might include vulnerabilities introduced in three ways [56, 57]. **(i)** The operating system installed in the container image can have vulnerabilities caused by missing security patches or outdated packages. **(ii)** The base image used to build the container image can introduce vulnerabilities from both the operating system and pre-installed applications or libraries. **(iii)** The applications installed or internally developed by the developers can introduce vulnerabilities.

Container image vulnerability scanners aim to identify the vulnerabilities inside a container image. Existing container image scanners are mainly software version-based, e.g., Clair [34], Trivy [35], Grype [36], and Stools [58]. That is, they unpack the container image, identify the installed software along with their versions, match them against publicly available vulnerability databases, e.g., National Vulnerability Database (NVD) [59], and list the reported vulnerabilities for that specific software. Therefore, they are able to identify the reported vulnerabilities of known software, but not unknown software. Though several scanners can support dynamic analysis, e.g., Trivy [35], they are not scalable for the HPC infrastructure.

Container runtime security. Containers have raised a number of security and privacy concerns, mainly because containers need to share the same kernel with the host. Extensive research has demonstrated that the approach, used by most existing containers (e.g., Docker [5], Singularity [40], Charliecloud [41], and Shifter [42]), is vulnerable to many cyber attacks, including privilege escalation [16, 60, 17, 61, 24], information leakage [62, 63, 29], resource exhaustion [30, 31, 33], and denial-of-service attacks [32]. Instead, many sandboxed runtimes have been developed to provide better isolation. Google has developed gVisor [64] as a middleware running between containers and the host kernel by implementing a large portion of the syscall interface with a smaller number of syscalls. Kata [65] from Intel and Firecracker [37] from Amazon use lightweight virtual machines as the container runtime. However, these solutions are not designed for HPC infrastructure as they cannot achieve satisfying performance for HPC applications [38].

4 Thrust 1: Efficient Vulnerability Detection for Container Images

To verify the security of the uploaded container images, ***this thrust aims to design an efficient image vulnerability scanner to accurately and quickly detect the vulnerabilities in the container images uploaded to the HPC infrastructure.***

Motivation. Existing software version-based container image scanners are able to identify the reported vulnerabilities in known software [34, 35, 36, 58], but they face a major challenge of **low coverage** due to two reasons. **(i)** The software version-based methods can not detect *unknown software*. That means, any

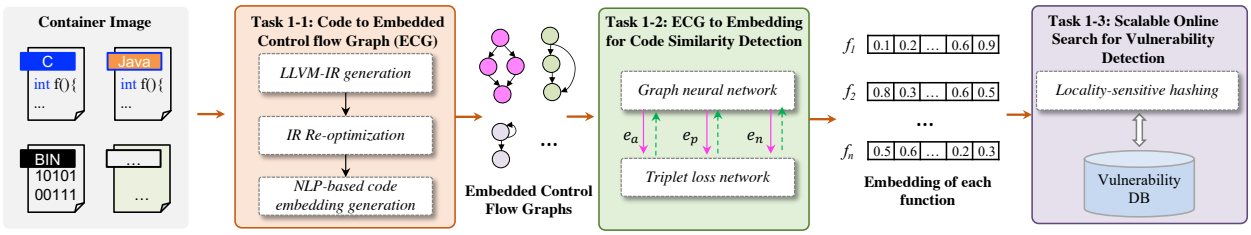

Figure 2: Overview of the proposed cross-language container image vulnerability scanner (**Thrust 1**). Given a container image, **Task 1-1** represents different types of code as embedded control flow graphs (ECGs), **Task 1-2** further converts an ECG to one embedding towards code similarity detection, and **Task 1-3** designs a scalable online search method against a pre-built vulnerability database.

software or code that is internally developed will not be detected. What is worse, in the HPC infrastructure, the scientists frequently develop such internal software, which shows a huge gap between the existing image scanner and the actual requirements. **(ii)** They can not detect the *newly updated software* as they have not been investigated properly and reported to the public vulnerability databases. This can miss the vulnerabilities existing in previous versions because they might still exist in the new versions.

Code similarity-based methods are shown to be capable of improving the coverage of vulnerability detection [66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81]. Their methodology is, if an unknown code is similar to a vulnerable code, it is highly possible to share the same vulnerability. As they do not rely on software versions, they are able to address the low coverage challenge caused by both unknown software and newly updated software. However, existing methods can not be directly applied to the container image scenario as **they are usually limited to one code type**, e.g., C source code, or C-compiled binary code. Differently, in a container image, there exists **a wide mix of various types of code**, e.g., C/C++, Java, Python, and binary code compiled from various programming languages.

Proposed research. To address this challenge, we propose a novel cross-language code similarity-based method for container image vulnerability detection as shown in Figure 2. First, we propose a unified code representation that is able to accommodate various types of programming languages (Task 1-1). With intermediate representation (IR) re-optimization and natural language processing (NLP)-based code embedding generation, the code is converted to the embedded control flow graph (ECG). Second, we propose an efficient code similarity detection method using graph neural networks and triplet loss network tailored for the unified code representation proposed in Task 1-1 (Task 1-2). With that, each function of the code is converted to an embedding. Third, we propose a scalable online search method for detecting vulnerabilities (Task 1-3). To this end, the code matching the vulnerabilities in the database will be outputted.

4.1 Task 1-1: ECG–A Language Agnostic Code Representation

The major challenge of cross-language code similarity detection is the different syntax and semantics between different programming languages. To address that, ***this task proposes a language-agnostic code representation, embedded control flow graph (ECG), to unify all of them into the same representation.***

Unified intermediate representation (IR). Figure 2 shows the process of generating the embedded control flow graph (ECG). First, given different types of code, we convert them to the same intermediate representation (IR), which is a language-agnostic code representation frequently used in compilers. In particular, an IR can be translated from different front-end programming languages and translated to different back-end architectures. Various IRs have been proposed in the past, e.g., LLVM-IR [82], Vex-IR [83], SPIR-V [84], and MLIR [85]. However, not all of them can be directly applied to the container image because it needs to support not only various programming languages but also binary code. Therefore, in this project, we propose to leverage LLVM-IR for two reasons. **(i)** LLVM-IR is the IR used in the LLVM compiler, which is widely adopted by the industry. LLVM-IR can support various programming languages as front-end, e.g.,

C, C++, C#, Fortran, Java, Matlab, and Python [82, 86, 87, 88], which has a comprehensive coverage of the programming languages used in scientific computing. **(ii)** More importantly, the binary code can also be converted to LLVM-IR with the help of decompilers, e.g., RetDec [89], McSema [90], and llvm-mctoll [91].

IR re-optimization. Though converting to the same IR helps to smooth the difference across various code types, such raw IRs are still dramatically different even for the source code with the same semantics but implemented with different programming languages. This is mainly derived from the following two challenges. **(i)** *Challenge #1: Binary code changes caused by compilation optimizations.* During the compilation process, the compiler will apply various optimization rules for better execution time, storage size, or power consumption. Many of the rules can dramatically change the code, e.g., `loop unrolling`, `tree vectorization`, and `function inlining` [81, 92]. It has become a major challenge for *binary-binary* or *source-binary* code similarity detection [81, 75, 77]. (ii) *Challenge #2: Syntax difference across different programming languages.* Though the raw IRs unified different source languages, the essential syntax difference between different languages will lead to dramatically different raw IRs.

Therefore, we propose to *re-optimize the raw IRs with the goal of minimizing the differences caused by compilation or different programming languages.* We propose to use a superset of the rules covered by the commonly used optimization levels, i.e., O1, O2, and O3, and exclude the conflict rules. We did not use a specific optimization level because the rules covered in different levels are quite diverse. For example, in LLVM (version 8.0.0) [93], O2 covers 83 rules, and O3 covers 86, while only 46% of the rules are covered by both. Such a re-optimization method can not only eliminate the differences introduced by the compilation process but also some noises across different programming languages. For example, the `dead code elimination` rule can help to remove the noisy code as it will never be executed.

Embedded control flow graph (ECG). With the re-optimized IR, we further represent it as a control flow graph (CFG), where a node denotes a basic block with the code executing together without interruption, and an edge denotes the control flow relationship between two nodes. CFG is frequently used for code analysis as it keeps the essential semantic information [75]. However, only using the CFG is insufficient as it ignores the syntax information. To accommodate that, we propose to *convert the statements in a basic block to an embedding* with natural language processing (NLP) techniques. This idea is feasible as the basic block embedding task is similar to the paragraph embedding task in NLP [94, 95, 96]. Basically, one line of IR code including multiple operands and operators is like a short sentence with a few words, and a basic block formed by multiple lines of IR code is like a paragraph. Motivated by that, we propose to apply Transformer models [97], e.g., BERT [96] for this task. Compared to other NLP models, Transformer models are effective for paragraph embedding because they are able to capture contextual information, bidirectional information, and fine-tune over a pre-trained model. However, a key challenge here is how to train an accurate model, which usually requires a large corpus. To achieve that, we first get a publicly available pre-trained code model, e.g., CodeBERT [98]. Then, we collect a corpus with re-optimized IRs from different source languages and binary code (details in Section 6). After that, we retrain the code models on our corpus so that we can get an accurate basic block embedding model.

PIs' preliminary results. We demonstrate the feasibility of using property graphs to represent the code for code similarity and vulnerability detection. In particular, in our work BugGraph [81], we use the control flow graph with eight manually defined features to represent a basic block. In our work Vestige [99], we represent a function as a feature vector with frequent control flow graph patterns and normalized instruction patterns. However, these works are limited to manually defined code features or a small subset of features and only focus on a single programming language. In this project, we plan to extend to multiple programming languages with a more generic code representation.

4.2 Task 1-2: Efficient Code Similarity Detection for Embedded Control Flow Graph

With the embedded control flow graph (ECG), the problem of code similarity has become ECG similarity. To accurately and efficiently compute that, ***this task proposes to leverage the graph triplet-loss network to compute the similarity between ECGs.***

Graph triplet-loss network for ECG similarity. At the heart of graph triplet-loss network (GTN) [81] is a graph neural network (GNN) model [100, 101, 102, 103], which can take an ECG as input and represent it as an embedding. However, the GNN model is insufficient by itself as it needs a proper loss function to supervise its learning process toward similarity computation. Therefore, we propose to use the triplet-loss network [104], which is used for face similarity computation. It takes as input a triplet of ECG embedding, i.e., $\{e_a, e_p, e_n\}$, aiming to maximize the similarity between the positive pair e_a and e_p, and minimize the similarity between the negative pair e_a and e_n. Formally, the loss value $\mathcal{L}$ is defined in Equation 1.

$$\mathcal{L} = \max\{sim(e_a, e_n) - sim(e_a, e_p) + \Delta, 0\} \tag{1}$$

The $sim()$ function calculates the similarity between two embedding, and Δ is a tunable margin value. The loss value $\mathcal{L}$ can be backpropagated to the GNN model so that we can use an optimizer (e.g., gradient optimization) to tune the trainable parameters of the GNN model toward minimizing the loss value. To this end, the trained GNN model can generate a representative embedding for similarity computation.

PIs' preliminary results. We demonstrate the feasibility of using graph triplet-loss network (GTN) for source-binary code similarity detection in our work BugGraph [81]. This preliminary work focused primarily on the similarity between the C language's source and binary code. Also, the attributes on the control flow graphs are manually defined. In addition, the similarity is limited to syntax same or similar code. In this project, we plan to extend this work to multiple languages with the help of embedded control flow graphs (Task 1-1), and also extend to semantic similar code.

Triplet generation. The key challenge of training an effective GTN model is to generate representative triplets covering all the similarity cases crossing the source and binary code of multiple programming languages. Given a large corpus of ECGs (the details of collecting them are in Section 6), we observe the triplets can be classified into two specific cases, i.e., the same language, and different languages.

(i) Within the same language, there are three types of similar pairs: *source-source*, *source-binary*, and *binary-binary*. The *source-source* pair represents two similar implementations. The *source-binary* pair represents that a source code is similar to the compiled binary. The *binary-binary* pair represents the binary code from the same source code is similar. For each similar pair, we need to find a different code to make up a triplet. We propose to randomly find a different source code for *source-source*, and a different binary code for *source-binary* and *binary-binary*.

(ii) Among two different languages a and b, there are four types of similar pairs: $source_a$-$source_b$, $source_a$-bin_b, bin_a-$source_b$, and bin_a-bin_b. The $source_a$-$source_b$ pair represents two similar implementations but from different languages. The $source_a$-bin_b pair represents that the source code from language a is similar to the binary code compiled from the similar source code b. The bin_a-$source_b$ pair represents that the binary code compiled from $source_a$ is similar to $source_b$. The bin_a-bin_b pair represents that the binary code compiled from $source_a$ is similar to the binary code compiled from $source_b$. To generate a triplet, for each similar pair, we randomly find a different source code from language b for $source_a$-$source_b$ and bin_a-$source_b$, and a different binary code from language b for $source_a$-bin_b and bin_a-bin_b.

However, simply generating all the triplets in this way will create an extremely large number of triplets, which will lead to ridiculously slow convergence. In particular, the case for different languages will dominate the number of triplets as it needs to combine any two languages in the corpus. To address this problem, we propose a two-step training method that can greatly reduce the triplets for different languages.

Two-step training. In the first step, we train a GTN model for the same language case with the triplets

only within the same language. Once the GTN model is converged, in the second step, we retrain it for the case of two different languages. *The key intuition is the GTN model converged for the same language is able to identify a source code and its compiled binary code as similar.* Therefore, in the second step, we only need to use one similar pair, i.e., $source_a$-$source_b$, while the other three similar pairs can be covered. This can reduce the number of generated triplets for different languages by more than 75% as one source code could be compiled into multiple binary code. However, instead of completely ignoring the other three similar pairs, we propose to still include a reasonable subset of them to boost the accuracy.

4.3 Task 1-3: Scalable Online Search for Vulnerability Detection

After Task 1-1 and Task 1-2, each code is represented as an embedding. Then, we can easily compute the similarity of two code as embedding similarity. With that, given an unknown code, we can compare it with the vulnerability from a pre-built vulnerability database. If it is highly similar to a vulnerable code, it might share the same vulnerability. The search for an unknown code against a vulnerability database is regarded as the online phase. As a vulnerability database may include a large number of samples, denoted as $\boldsymbol{n}$, and a container image can also involve an excessive amount of code, denoted as $\boldsymbol{m}$, the online code similarity detection problem is actually an ***m-to-n search problem***. Therefore, we need a scalable solution as scientific computing applications are usually time-sensitive.

Motivated by that, ***this task aims to design a scalable online search solution for vulnerability detection***. In particular, we first propose a locality-sensitive hashing-based online search method. Then, we propose a hybrid solution for vulnerability detection between code similarity and software version-based.

Locality-sensitive hashing (LSH)-based online search. The goal of online search is to quickly find the code embedding in a large database that is closest to a queried embedding, which is a classical nearest neighbor search problem [105]. To solve it, we propose to leverage the scalable locality-sensitive hashing (LSH) [106, 107] technique. We choose LSH because it is not only fast but also space efficient, while other solutions, e.g., Voronoi diagram [108] and KD-tree [109], either have a high search time complexity close to $\mathcal{O}(n)$ or an exploding space complexity that is close to the exponential with the embedding dimension, i.e., $\mathcal{O}(n^d)$, where n and d denote the size of the database and embedding dimension, respectively. In particular, LSH hashes similar input data points into the same *bucket* with high probabilities. Different from conventional hashing techniques that aim to minimize hash collisions, LSH aims to maximize them. LSH learns a projection function $f()$ so that if two data points are closer in the high-dimensional embedding space, they should remain close after the projection in the hashing space. The closer the two points are, the more similar they are. For any code embedding e_a, e_b that are close to each other in the high-dimensional embedding space, there is a high probability $P[f(e_a) = f(e_b)] = p_1$ that they fall into the same bucket. Likewise, for any code embedding that is far apart, there is a low probability $p_2 (p_2 < p_1)$ that they fall into the same bucket. With LSH, the large vulnerability database is converted to an LSH hashing database. Then, during the online search for a new data point, LSH can quickly find a bucket of data points that are similar with high probability. Though the actual time and space complexity are decided by the algorithm and actual parameters, they are usually very efficient. For example, an LSH method is able to achieve $\mathcal{O}(n^{0.25})$ online search time with $\mathcal{O}(n^{1.25})$ space [110].

Though LSH-based online search is able to efficiently compute one query, only using it is still inefficient for vulnerability detection as there might exist a large number of code in a container image to be searched, e.g., $\boldsymbol{m}$. Therefore, we propose an efficient hybrid solution for vulnerability detection. The key insight is *we can quickly filter out the code that can be easily detected by software version-based methods so that the number of code to be searched online can be greatly reduced.*

Hybrid vulnerability detection. For vulnerability detection, though the software version-based method faces a major problem of low coverage for unknown software or newly updated software, it has two advantages. First, it is very fast as it simply queries a software version against a vulnerability database without any

complex computation. Second, the reported vulnerabilities are true positives that are usually confirmed by the vendors. Motivated by that, we propose to first apply software version-based image scanners to quickly identify the vulnerabilities in known software. However, important discrepancies are found between different image scanners. For example, a recent study [15] on scientific container images shows that only 60% of the detected vulnerabilities are shared among two popular image scanners, i.e., Anchore [36] and Clair [34]. Therefore, we propose to leverage a basket of existing image scanners to have better coverage of known software. We plan to use three image scanners that show significant discrepancies. For the remaining code, we will apply an LSH-based online search method against a vulnerability database, which will be built offline with the knowledge from publicly disclosed vulnerabilities, e.g., Common Vulnerabilities and Exposures (CVE) [111] and National Vulnerability Database (NVD) [59].

PIs' preliminary results. In our preliminary work, BugGraph [81], we have built a vulnerability database with over 200 known vulnerabilities collected from CVE [111]. We also applied it to firmware vulnerability detection. We identified 140 vulnerabilities in six commercial firmware. In this project, we plan to further expand our vulnerability database, design an LSH-based code similarity detection method and integrate with several software version-based image scanners.

5 Thrust 2: Secure, Lightweight and High-Performance Container Runtime

Though the newly designed image vulnerability scanner (Thrust 1) is able to verify the security of the uploaded container image, it can not identify the security threats of the container runtime running in the high-performance computing (HPC) infrastructure. Therefore, ***this thrust aims to design a container runtime tailored for the HPC infrastructure, which not only is secure but also maintains high performance.***

Motivation. Container runtime is responsible for running containers and supporting containerized processes. It translates the visibility and resource restrictions from the user-facing API into directives for the kernel. Docker utilizes a *native (i.e., bare-metal) runtime*, i.e., runC [112], which only setups the necessary environment before the container process starts, allowing the container to share the host kernel directly. Existing HPC runtimes (e.g., Singularity [40], Charliecloud [41], and Shifter [42]) also fall into this category. Bare-metal runtimes achieve isolation with Linux kernel features, e.g., namespaces, cgroups, SELinux, capabilities, and AppArmor.

However, the bare-metal runtime can lead to **container escape vulnerabilities** that containers can escalate their privileges [16, 60, 17, 61, 24]. Besides container escape, the bare-metal runtime is vulnerable to other vulnerabilities under multi-tenant scenarios, as one tenant can potentially affect others with shared hardware resources. Table 2 summarizes them with our *preliminary results* shaded, including information leakage [62, 63, 29], resource exhaustion [30, 31], virtual resource denial-of-service [32], memory ill-allocation [33], and performance degradation [113]. Among the vulnerabilities, we observe some can dramatically lower the performance, which is critical in HPC infrastructure. E.g., in our preliminary work [30], we find the performance of a server can drop by 95% with the existence of only one unprivileged malicious container using limited resources (e.g., 50% CPU utilization of one CPU core).

Table 2: Other security vulnerabilities in containers. Our preliminary results are shaded.

Issues	Impact
CPU cgroup break [30]	95% degradation with `sysbench`
Memory cgroup break [33]	>16GB extra memory consumed
Virtual resource contention [32]	97.3% degradation with `dd`
CFS issues [113]	5x longer for `steamcluster`
Information leakage [62]	Expose all processes information
Global power leakage [63, 29]	Advanced cloud attacks
Race condition [31]	Container crash
Thread oversubscription [114]	25x performance slowdown
Network softirq overhead [115]	6x slowdown with `sysbench`

Different from the bare-metal runtime, existing cloud vendors seek another solution, i.e., *sandboxed runtime*, which introduces a translation layer between the container and the host kernel. For example, Google develops gVisor [64], a middleware that can reduce the attack surface of the host kernel by implementing a large portion of the syscall interface with a smaller number of syscalls. Intel develops Kata [65] to use

lightweight virtual machines as the container runtime. Similarly, Amazon AWS develops Firecracker [37] which is specifically designed for serverless computing. However, the sandboxed runtime faces **performance issues** when being deployed to the HPC infrastructure. In particular, though gVisor and the obsoleted Kata 1.0 support the integration with Docker, they suffer from low performance for HPC workloads [38]. Firecracker is optimized for lightweight functions that can finish within seconds, which is not suitable for complex HPC workloads as they usually take much longer time [39].

Proposed research. We propose to design a secure, lightweight, and high-performance runtime, HPContainer, to run containers on HPC infrastructure. The runtime is transparent to HPC users, i.e., users can still run Docker or Singularity images as before. Meanwhile, the runtime enables high-level isolation with minimal overhead, while still maintaining high performance. In particular, HPContainer adopts multiple heuristics to specifically optimize the performance for HPC workloads (Task 2-1). It utilizes the Linux Kernel's KVM (Kernel-based VM) virtualization infrastructure to spawn either optimized QEMU (Quick EMUlator) VMs or provide minimal VMs (Task 2-2). Further, it dynamically debloats the unnecessary files in the container image to further reduce the attack surfaces (Task 2-3).

Figure 3 presents the overall architecture of HPContainer. We reuse the Docker container engine (i.e., dockerd) to handle high-level tasks so that HPContainer can seamlessly integrate with Docker CLI (Command Line Interface). For each created container, a long-lived shim process with the same life cycle as the container will be developed to abstract the low-level runtime, intercept/redirect the container's data streams, and track/synchronize container status. The HPContainer runtime will spawn and run containers with predefined resources, e.g., the number of CPUs, NUMA nodes, and GPUs.

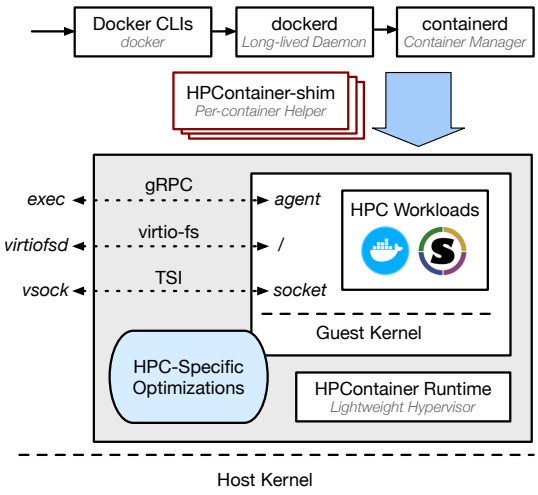

Figure 3: Overview of the proposed HPContainer.

To this end, the HPContainer serves as a hypervisor, and the spawned container runs inside a lightweight virtual machine (VM), where an agent process communicates with the runtime via gRPC and acts as the supervisor to manage the workload. Additionally, outside the VM, we further implement a sandbox to protect the host server from unwanted malicious behaviors against the hypervisor, e.g., vulnerabilities that allow the guest to inject code. This level of protection places the VM into a restrictive sandbox, with multiple container security mechanisms adopted, e.g., chroot, namespaces, capabilities, and seccomp.

Preliminary results. The PIs have conducted in-depth research on disclosing security vulnerabilities in Linux containers, including information leakage problems due to the incompleteness in namespaces [62, 63, 29], as well as performance-related issues in Linux control groups (e.g., cpu, cpuset, memory subsystems) [30, 31]. Also, the PIs have uncovered several security vulnerabilities in VM migrations [116, 117]. Meanwhile, the PIs have experience in building systems, including several VM migration mechanisms based on KVM/QEMU [118, 119] and Linux kernel components (filed as a US patent) [120].

5.1 Task 2-1: Containerized Performance Optimization Tailored for HPC Infrastructure

The HPC infrastructure employs powerful servers usually with multiple processor sockets and NUMA (non-uniform memory access) nodes. For example, the Intel Xeon Platinum system has 12 processor sockets, with a total of 288 cores in a multi-tier NUMA hierarchy. Unfortunately, the deployment and layout of hardware components cause hard performance penalties that cannot be easily mitigated. For example, the memory latency is not uniform between NUMA nodes because it will increase by 60-70% with each additional hop outside of a NUMA node [121]. However, such penalties can be largely reduced by a well-designed software

scheduler to make a smart placement of workloads, e.g., to schedule guest workloads running inside a node. Since HPContainer adds a virtualized layer between the host and containers (e.g., containers actually see vCPU instead of physical CPU cores), it also provides HPContainer an opportunity to optimize resource management and accelerate performance for HPC workloads. Therefore, ***this task proposes an adaptive strategy to manage the virtual resources and several heuristics to reduce the system latency.***

Adaptive virtual resource management. We propose to adaptively manage two virtual resources, i.e., hyper-threading and caches, which are important for runtime performance.

- **Hyper-threading** (a.k.a, simultaneous multithreading) [122] is a technique for improving the overall efficiency of Intel CPUs with hardware multithreading. It splits a physical core into two or more virtual cores, which share the same resources, e.g., cache, and registers. As less idle time is spent by the internal microarchitecture components, two logical cores are more efficient than a traditional single-threaded core. Due to that, Intel claims hyperthreading achieves a 15%-30% performance boost. However, the performance boost may not apply to individual programs, as the shared resource also introduces contentions that will jeopardize the performance. For example, with two vCPUs scheduled on the same physical core, the overall throughput might be boosted to 130%, but each individual vCPU actually has a reduced throughput (e.g., 80%), compared with scheduling on two physical cores.
- The **cache** is important for the runtime performance as it stores the frequently accessed data in a small but high-speed memory so that it can avoid repeated, slower accesses to the system's main memory or storage. L1/L2 cache is shared by a core, while the last-level cache (LLC) is shared among cores on the same CPU socket. If two vCPUs are scheduled to two different sockets, the performance of the computing job will be decreased due to the high cache (L1, L2, and LLC) miss rate.

Motivated by that, we propose an adaptive strategy to manage virtual resources. In particular, if the system is not busy, which can be measured by the low CPU utilization, HPContainer will try not to schedule two vCPUs on the same physical core to avoid resource contentions. Meanwhile, HPContainer attempts to schedule the vCPUs to the same CPU socket, so that they can enjoy the benefits of the fast cache. On the other hand, if the system is busy when hyperthreading is unavoidable, which can boost the overall performance, HPContainer will try to schedule two vCPUs of the same guest VM on the same physical core, so that they can benefit from the low-level caches, e.g., L1 and L2.

Reducing system latency. To make the performance close to running on bare-metal hardware, we further propose to reduce the system latency from three levels, i.e., thread, memory, and network.

- For **threads**, we propose to utilize the `cpuset` to set the affinity of threads. The `cpuset` is a subsystem of the Linux Control Groups (cgroups) feature that can assign specific CPUs and memory nodes to a group of processes. Previous work [121] has shown that the performance degrades significantly for HPC workloads without thread pinning, especially when the application crosses the NUMA group boundaries. The reason is that threads are migrated outside of their original NUMA node and group, resulting in increased memory latency.
- For **memory**, we propose to ensure all the pages for the guest container are already mapped before the container starts. As HPC infrastructure has abundant resources, many mechanisms will be disabled, including memory remap and memory fragmentation so that the allocated memory for the container will not be moved. The idea is to enforce full reservation of memory (not only memory allocation) for the container so that the performance can be guaranteed.
- For the **network**, we propose to disable the large receive offload (LRO) which is supposed to improve the performance of network data transmission by aggregating multiple small packets into a larger, single packet before it is processed by the system. As the jobs running on the HPC infrastructure typically do not incur huge network traffic, we can disable the LRO to reduce the latency. Also, we propose to enable Virtual Network Interrupt (vNIC) coalescing to reduce the number of interrupts by allowing the system to combine multiple interrupt requests from the NIC into one.

5.2 Task 2-2: Customized Lightweight and Secure Hypervisor for HPC Infrastructure

HPContainer is a hypervisor (a.k.a, virtual machine monitor, VMM)-based container runtime. Conventional hypervisors, e.g., KVM [123], and QEMU [124], mainly aim to provide complete coverage of all the features, while many of them are unnecessary in the HPC infrastructure, e.g., USB. Though Firecracker [37] provides a lightweight hypervisor, it is not suitable for HPC infrastructure as it neither is compatible with Docker nor removes many HPC-required low-level features, e.g., GPU support. Therefore, ***this task aims to design a customized hypervisor that is not only lightweight but also secure for HPC infrastructure***.

We propose to develop the hypervisor based on an open-source project, i.e., rust-vmm [125], which is a Rust-based hypervisor. We choose it because Rust is a memory-safe programming language, which can provide a secure foundation for building reliable and efficient hypervisors. Further, we propose to build on top of existing Linux components (e.g., process scheduler and memory manager) as Linux has proven to be efficient for running multi-tenant workloads on multi-processor machines. Also, we propose to customize the hypervisor from the following four levels, i.e., hypervisor features, file system, network, and graphics processing unit (GPU).

- **Hypervisor features.** We propose to remove as many unnecessary features as possible because they not only hurt the performance but also provide more attack surfaces. We plan to take a deep study of all the required features in HPC infrastructure and remove the remaining. As a start, we will remove the following features, such as offering BIOS, booting arbitrary kernels, emulating legacy devices (e.g., USB and sound), and supporting live VM migration.
- **File system.** As most HPC workloads are I/O (input/output) intensive, we need to consider the trade-off between security isolation and I/O performance. Motivated by that, we will explore virtio-fs [126], a shared file system that lets VMs access a directory tree on the host. This approach enables the hypervisor to use directories in the host as the guest's root file system, which can achieve close performance as the bare-metal runtime [127]. Besides, it can minimize storage management efforts, and thus is friendly to HPC workloads. Further, HPContainer adopts an additional level of protection by placing the VM into a restrictive sandbox with namespace enabled. This mechanism can hide storage systems from VMs. Guests have root access to the shared directory, but cannot access other files on the host.
- **Network.** To avoid the complex virtual bridge setup for enabling networks inside VMs, we will explore the Transparent Socket Impersonation (TSI) [128] with VM Sockets (e.g., virtio-vsock [129]) for networking. The VM Sockets address family facilitates communication between the host and VMs, and is independent of virtual machine network configuration [130]. TSI basically bridges all sockets (e.g., `AF_INET`) in the guest to `AF_VSOCK` when communicating outside the VM. This allows a VM to have network connections without virtual interfaces, providing both inbound and outbound networking capabilities with near-zero configuration efforts. Further, nontrivial human efforts of implementing a TCP network stack can be saved.
- **GPU.** Finally, we will explore the NVIDIA Multi-Instance GPU (MIG) feature to isolate GPU usage, which is becoming increasingly important for scientific computation. Compared with vGPU which enables logical GPU virtualization and GPU access to multiple clients on a single hypervisor [131, 132], MIG is a mechanism available on recent GPUs that enhances vGPU by enabling spatially partitioned GPU instances [133]. MIG partitions GPU compute resources (e.g., GPU engines) into GPU instances, and each GPU instance has separate and isolated paths through the entire memory system. The on-chip crossbar ports, memory controllers, low-level cache banks, and DRAM buses are all assigned uniquely to an individual instance. Such strong isolation provides both security and predictable performance guarantees to users of shared GPU. With MIG, users will be able to see and schedule tasks on the GPU Instances as if they were standalone GPUs. In HPContainer, each container instance will get an isolated GPU instance enabled by MIG.

5.3 Task 2-3: Reducing Image Attack Surfaces with Dynamic Debloating

A container image might involve unnecessary software, which provides additional attack surfaces. In a recent study of real-world neuroscience container images [15], unnecessary software can contribute up to 70% of all the detected vulnerabilities. What is more, as the workloads running in the HPC infrastructure have more restrictions, the unnecessary software and files are even more. For example, debugging tools, administration tools, monitoring tools, and inspection tools, are usually not allowed to run in the HPC infrastructure. To further reduce the attack surfaces, ***this task aims to design a container runtime that can dynamically debloat the images***.

Given a container image, we run it in our proposed HPContainer with small-size inputs. We will add a custom system call tracer to record all the executed software and files. We also leverage the image scanners used in Task 1-3, e.g., Trivy [35], and Clair [34], to identify unnecessary software and files. The image scanners use both software metadata and dependencies to determine if the software is actually used or not. Then, we dynamically build a new minimal VM image with only the necessary software and files. However, there is a caveat that the unnecessary files may be identified mistakenly, especially by the dynamic running with small-size inputs. Therefore, we propose to leverage the overlay tools like VMSH [134] to attach the needed software via side-loading kernel code from the hypervisor into the guest.

6 Task 3: System Integration, Dataset, and Evaluation

System integration. The PIs plan to conduct the system integration with high-performance computing (HPC) stakeholders. We have initiated conversations with administrates from the University of North Texas (UNT) HPC Services, Texas Advanced Computing Center (TACC), and National Oceanic and Atmospheric Administration (NOAA). Though we did not get letters of collaboration due to the complex administrative process, they expressed strong interest as container security has become a serious concern for them. In fact, UNT is one of the institutes to fund TACC Lonestar6 [135], the newest HPC system of TACC's Lonestar series. Senior personnel Herrington, an advanced computing services consultant at UNT, is the main point of contact for TACC from the UNT side and has recurring meetings with the TACC administrative team. We are also in the process of communicating with other HPC teams, e.g., Jetstream2 [136] and Chameleon [137].

The proposed container image scanner will be deployed on a regular computing node in an HPC cluster to scan any uploaded container image. As a submitted job usually has to wait for the required resources, the image scanner can leverage this waiting period to scan it, which will not cause extra waiting time for the job. The proposed system will be implemented on top of the latest Linux kernel with the container engine so that existing HPC software is compatible. The container runtime will support real-world scientific usage with all mechanisms enabled (e.g., *cgroups, namespace, seccomp, and capabilities*.). For fast prototyping, the development will be first conducted on internal servers and moved to the HPC clusters later.

Dataset. We will collect three types of data as detailed below. **(i)** We will collect a large corpus of similar code crossing various programming languages. Currently, we have collected a total of over 1M source code functions as shown in Table 3, crossing five different programming languages, including C, C++, C#, Java, and Python. They can further be compiled into binary code with various compilation options. In this project, we will continue to collect more code similarity datasets with the goal of covering as many languages used in scientific applications as possible. **(ii)** We will create a large vulnerability database (DB). Currently, we have built a vulnerability DB with over 200 known vulnerabilities collected from CVE [111]. We plan to scale up the vulnerability DB by crawling more known vulnerabilities from public resources, e.g., CVE [111], and

Table 3: Currently collected code similarity corpus.

Dataset	# Functions	Language
CLCDSA [138]	30K	C++, C#, Java, Python
POJ-104 [139]	50k	C, C++
AtCoder [140]	90k	Java, Python
Source-binary [81]	100k	C, C++
BigCloneBench [141]	765k	Java
Total	>1M	C, C++, C#, Java, Python

NVD [59]. **(iii)** We will collect a large number of scientific container images with the goal of covering various domains, e.g., biology, physics, and chemistry. We will also collect some large container images to test the scalability of our proposed method.

Evaluation. This project will be evaluated for both the image vulnerability scanner and container runtime.

Image vulnerability scanner evaluation: We plan to evaluate it (Thrust 1) from three aspects. **(i)** We will evaluate the code similarity detection method with the code corpus dataset using the metric using true positive rate (TPR), false positive rate (FPR), true negative rate (TNR), false negative rate (FNR), accuracy, and top-k hit rate. **(ii)** We will evaluate the code similarity detection overhead in terms of run time for both training and online search. **(iii)** We will evaluate vulnerability detection against the containers having known vulnerabilities with the metrics of TPR, FPR, TNR, FNR, accuracy, and top-k hit rate. We will compare it with various code similarity detection methods and conventional image scanners.

Container runtime evaluation: We plan to evaluate it (Thrust 2) on the collected container images from both performance and security. **(i)** For performance evaluation, we will report various performance statistics besides the absolute runtime, e.g., the number of instruction cycles, cache miss rate, memory bandwidth usage, power consumption, throughput, and hardware utilization. We will compare with vanilla bare-metal Docker/Singularity as well as other sandboxed container runtimes. **(ii)** For security evaluation, we will emulate a practical real-world scenario in an HPC cluster where attackers can control one or multiple unprivileged containers with limited resource restrictions. Then we will evaluate all security vulnerabilities (that are still available to the existing containers) mentioned in Thrust 2, e.g., container escape [16, 60, 17, 61, 24], information leakage [62, 63, 29], and resource exhaustion [30, 31]. We will measure the impact on both individual containers (e.g., belonging to different tenants) running different workloads and overall HPC cluster performance, and compare HPContainer with existing solutions (e.g., Docker and Singularity).

Timeline. Table 4 outlines a three-year timeline along with yearly goals for this project. PI Ji will mainly lead efforts in Thrust 1 and PI Gao will mainly lead efforts in Thrust 2. Both PIs will co-lead Task 3. PI Ji will be responsible for the overall project management including the scientific and administrative coordination of the project, fiscal management, quality control, and training and supervision of project staff. Meanwhile, both PIs will hold weekly Zoom meetings to synchronize progress. Though each task has a designated lead university, they will be conducted jointly by both teams.

Table 4: Project timeline.

Tasks	Milestones						Lead PI
	Year 1		Year 2		Year 3		
Task 1-1	✓	✓					Ji
Task 1-2			✓	✓			Ji
Task 1-3					✓	✓	Ji
Task 2-1	✓	✓					Gao
Task 2-2			✓	✓			Gao
Task 2-3					✓	✓	Gao
Task 3		✓		✓		✓	Ji & Gao

7 Broader Impacts

Societal impact. The proposed research, once successful, will significantly advance the security of containers in high-performance computing (HPC) infrastructures. We will share our results and findings via academic publications to broadly disseminate our research to the scientific computing, Cybersecurity, and HPC research communities. We will release our tools and systems as open-source software to help HPC practitioners and domain scientific computation developers to protect their computation systems, sensitive data, and internally developed software. In the long term, we anticipate that our work will advance general-purpose container design toward enabling more secure and efficient computation.

Education programs and curriculum development. Efforts from this project will be integrated toward establishing and improving security and HPC education programs at the PIs' home institutions. These programs include the B.S. in Cybersecurity and M.S. in Cybersecurity at the University of North Texas (UNT); and CS concentrations on HPC and Cybersecurity at the University of Delaware (UD). At UNT, PI Ji has taught CSCE 5565: Secure Software Systems, which is a core course for M.S. in Cybersecurity. PI Ji will develop several new course modules, e.g., container security, and secure container design, and incorporate

them into existing courses that he is teaching, e.g., CSCE 5565, and CSCE 6933: Advanced Topics in CS. PI Gao has taught four security courses at UD, including two newly designed computer security courses (e.g., CISC 469/669 Computer Security Principles and Practice), and has integrated new topics from system and networking security, including Linux container and Docker security. PI Gao has designed a new container security lab using Docker to teach virtualization/cloud computing in CISC 449/649 class at UD. Both PIs plan to keep this trend by integrating the results of this research into the existing curriculum.

Commitment to student training and diversity. Both UNT and UD have extensive undergraduate research programs. PI Ji has advised 6 female students (2 graduate students, 1 undergraduate, and 3 high school students) on HPC and container security. UNT is designated as a **Minority Serving Institution (MSI)** and a **Hispanic Serving Institution (HSI)**. PI Gao has mentored 2 female undergraduate students working on container security focusing on mobile and cloud computing environments. In this project, the PIs will further enhance the undergraduate research programs by recruiting undergraduate students into our research. The selected undergraduate students will spend a substantial amount of time closely working with graduate students. Further, both PIs will actively participate in the REU site supported by NSF. In fact, PI Ji has advised two students (one female) in summer 2022 supported by the REU site *TaMaLe (Testing and Machine Learning for Context-Driven Systems)*, and is advising one female student under the *CAHSI Local Research Experiences for Undergraduates (LREU)* program for spring 2023. Recently, PI JI is selected as a fellow of an NSF-HSI supported UNT Diversity & Excellence in Engineering Network (DEEN) program [142], where he will provide research and engineering experience to traditionally under-represented populations.

Outreach and educational impact. Our community outreach goal is to increase the overall awareness and knowledge of Cybersecurity and HPC in the states of Texas and Delaware, which is an **EPSCoR** state in great need of education and research activities. The PIs are dedicated to recruiting underrepresented students and promoting STEM education at the K-12 level. PI Ji is closely working with the Texas Academy of Mathematics & Science (TAMS) program hosted at UNT, which is the nation's first early college entrance residential program for gifted high school students. Currently, PI Ji is advising 6 TAMS students (3 female) on Cybersecurity and HPC research. In particular, two TAMS students, Lillian Wang (female) and Avik Malladi, published a poster at the prestigious HPC conference, SC'22 [143]. The CSE department at UNT is part of the BRAID (Building, Recruiting, And Inclusion for Diversity) initiative which focuses on training high school teachers in teaching computer science. Particularly, PI Ji organized a panel titled Recent Advances in Security and Privacy at **Texas CSTA Chapters Conference 2023** [144], to work with high school teachers to develop modules on security and privacy. Also, PI Ji gave a tutorial on Graph Algorithms at **Digital Divas 2023 to female high school students** [145]. PI Gao has recruited two female undergraduate students at UD. One of them has been admitted as a graduate student to Dartmouth College with scholarships. This project will provide much-needed financial support and research opportunities for these female and minority graduate students. PI Gao has experience hosting summer camps for high school students to promote Cyber education and is involved with multiple activities in the Cyber and Girls Experiencing Engineering camps. They will continue this trend as part of this project and advocate the inclusion of underrepresented students to pursue their careers in computer science and engineering.

8 Results from Prior NSF Support

PI Ji has no prior NSF support. **PI Gao** is the PI of the project: "CRII: SaTC: Securing Containers in Multi-Tenant Environment via Augmenting Linux Control Groups", Award Number: CNS-2054657, 2020/5/31 – 2023/5/31, $174,995. **Intellectual Merit**: Designed systems to uncover escaping strategies to break the resource rein of Linux control groups, and develop mechanisms to mitigate related attacks. **Broader Impacts**: This project has supported 1 undergraduate student and 1 Ph.D. student. The success of this project will advance state-of-the-art container design, e.g., Docker. **Results**: So far, papers resulting from the project include IEEE Security & Privacy [43], Usenix Security [46, 45], AsiaCCS [29], and DSN [117, 31].

References

[1] J. Michalakes, J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock, and W. Wang, "The weather research and forecast model: software architecture and performance," in *Use of high performance computing in meteorology*. World Scientific, 2005, pp. 156–168.

[2] J.-C. André, G. Aloisio, J. Biercamp, R. Budich, S. Joussaume, B. Lawrence, and S. Valcke, "High-performance computing for climate modeling," *Bulletin of the American Meteorological Society*, vol. 95, no. 5, pp. ES97–ES100, 2014.

[3] P. Angelikopoulos, C. Papadimitriou, and P. Koumoutsakos, "Bayesian uncertainty quantification and propagation in molecular dynamics simulations: a high performance computing framework," *The Journal of chemical physics*, vol. 137, no. 14, p. 144103, 2012.

[4] F. Shao, Z. Chang, and Y. Zhang, "Aes encryption algorithm based on the high performance computing of gpu," in *2010 Second International Conference on Communication Software and Networks*. IEEE, 2010, pp. 588–590.

[5] "Docker: Accelerated, Containerized Application," https://www.docker.com/.

[6] "Singularity Container Technology & Services | Sylabs," https://sylabs.io/.

[7] "Summit - Oak Ridge Leadership Computing Facility," https://www.olcf.ornl.gov/summit/.

[8] "Sierra | HPC @ LLNL," https://hpc.llnl.gov/hardware/compute-platforms/sierra.

[9] "Frontera," https://frontera-portal.tacc.utexas.edu/.

[10] S. Peisert, "Security in high-performance computing environments," *Communications of the ACM*, vol. 60, no. 9, pp. 72–80, 2017.

[11] R. C. Green, L. Wang, and M. Alam, "High performance computing for electric power systems: Applications and trends," in *2011 IEEE Power and Energy Society general meeting*. IEEE, 2011, pp. 1–8.

[12] "University of Utah update on data security incident," https://attheu.utah.edu/facultystaff/university-of-utah-update-on-data-security-incident/, 2020.

[13] "Carbanak, Anunak, Group G0008 | Mitre Att&CK," https://attack.mitre.org/groups/G0008/, 2020.

[14] "European supercomputers hacked in mysterious cyberattacks," https://www.bleepingcomputer.com/news/security/european-supercomputers-hacked-in-mysterious-cyberattacks/, 2020.

[15] B. Kaur, M. Dugré, A. Hanna, and T. Glatard, "An analysis of security vulnerabilities in container images for scientific data analysis," *GigaScience*, vol. 10, no. 6, p. giab025, 2021.

[16] "CVE-2022-0185 in Linux Kernel Can Allow Container Escape in Kubernetes," https://blog.aquasec.com/cve-2022-0185-linux-kernel-container-escape-in-kubernetes.

[17] "CVE-2022-0492: Privilege escalation vulnerability causing container escape," https://sysdig.com/blog/detecting-mitigating-cve-2022-0492-sysdig/.

[18] "CVE-2022-23648 Detail," https://nvd.nist.gov/vuln/detail/CVE-2022-23648.

[19] "CVE-2022-0847," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0847.

[20] "CVE-2021-30465 Detail," https://nvd.nist.gov/vuln/detail/CVE-2021-30465.

[21] "CVE-2021-22555," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-22555.

[22] "CVE-2021-31440 Detail," https://nvd.nist.gov/vuln/detail/CVE-2021-31440.

[23] "CVE-2020-8835 Detail," https://nvd.nist.gov/vuln/detail/CVE-2020-8835.

[24] "CVE-2019-5736: RunC Container Escape Vulnerability Provides Root Access to the Target Machine," https://www.trendmicro.com/vinfo/fr/security/news/vulnerabilities-and-exploits/cve-2019-5736-runc-container-escape-vulnerability-provides-root-access-to-the-target-machine.

[25] "CVE-2017-5123 Detail," https://nvd.nist.gov/vuln/detail/cve-2017-5123.

[26] "CVE-2017-1000112 Detail," https://nvd.nist.gov/vuln/detail/cve-2017-1000112.

[27] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A measurement study on linux container security: Attacks and countermeasures," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 418–429.

[28] X. Gao, B. Steenkamer, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "A Study on the Security Implications of Information Leakages in Container Clouds," *IEEE Transactions on Dependable and Secure Computing*, 2018.

[29] Z. Zhang, S. Liang, F. Yao, and X. Gao, "Red alert for power leakage: Exploiting intel rapl-induced side channels," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 162–175.

[30] X. Gao, Z. Gu, Z. Li, H. Jamjoom, and C. Wang, "Houdini's Escape: Breaking the Resource Rein of Linux Control Groups," in *ACM CCS*, 2019.

[31] K. McDonough, X. Gao, S. Wang, and H. Wang, "TORPEDO: A Fuzzing Framework for Discovering Adversarial Container Workloads," in *The IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2022.

[32] N. Yang, W. Shen, J. Li, Y. Yang, K. Lu, J. Xiao, T. Zhou, C. Qin, W. Yu, and J. Ma, "Demons in the shared kernel: Abstract resource attacks against os-level virtualization," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 764–778.

[33] Y. Yang, W. Shen, X. Xie, K. Lu, M. Wang, T. Zhou, C. Qin, W. Yu, and K. Ren, "Making memory account accountable: Analyzing and detecting memory missing-account bugs for container platforms," in *Annual Computer Security Applications Conference*, 2022, pp. 869–880.

[34] C. Team, "What is ClairV4 - Clair Documentation." [Online]. Available: https://quay.github.io/clair/

[35] A. Security, "Overview - Trivy." [Online]. Available: https://aquasecurity.github.io/trivy/v0.35/

[36] anchor, "grype: A vulnerability scanner for container images and filesystems." [Online]. Available: https://github.com/anchore/grype

[37] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, 2020, pp. 419–434.

[38] "I/O performance of Kata containers," https://www.stackhpc.com/kata-io-1.html.

[39] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[40] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PloS one*, vol. 12, no. 5, p. e0177459, 2017.

[41] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2017, pp. 1–10.

[42] L. Gerhardt, W. Bhimji, S. Canon, M. Fasel, D. Jacobsen, M. Mustafa, J. Porter, and V. Tsulaia, "Shifter: Containers for hpc," in *Journal of physics: Conference series*, vol. 898, no. 8. IOP Publishing, 2017, p. 082021.

[43] P. Cronin, X. Gao, H. Wang, and C. Cotton, "Time-Print: Authenticating USB Flash Drives with Novel Timing Fingerprints," in *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[44] Y. Gu, L. Ying, Y. Pu, X. Hu, H. Chai, R. Wang, X. Gao, and H. Duan, "Investigating package related security threats in software registries," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 1151–1168.

[45] P. Cronin, X. Gao, C. Yang, and H. Wang, "Charger-surfing: Exploiting a power line side-channel for smartphone information leakage," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[46] G. Liu, X. Gao, H. Wang, and K. Sun, "Exploring the Unchartered Space of Container Registry Typosquatting," in *USENIX Security Symposium (Security)*, 2022.

[47] Y. Ji, M. Elsabagh, R. Johnson, and A. Stavrou, "DEFInit: An Analysis of Exposed Android Init Routines," in *30th USENIX Security Symposium (USENIX Security)*, 2021.

[48] G. Liu, D. Liu, S. Hao, X. Gao, K. Sun, and H. Wang, "Ready raider one: Exploring the misuse of cloud gaming services," 2022.

[49] X. Gao, Z. Xu, H. Wang, L. Li, and X. Wang, "Reduced Cooling Redundancy: A New Security Vulnerability in a Hot Data Center," in *The Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[50] Y. Ji, H. Liu, and H. H. Huang, "iSpan: Parallel Identification of Strongly Connected Components with Spanning Trees," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2018, pp. 731–742.

[51] Y. Ji and H. H. Huang, "Aquila: Adaptive Parallel Computation of Graph Connectivity Queries," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2020.

[52] Q. Fu, Y. Ji, and H. H. Huang, "Tlpgnn: A lightweight two-level parallelism paradigm for graph neural network computation on gpu," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, pp. 122–134.

[53] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An Updated Performance Comparison of Virtual Machines and Linux Containers," in *IEEE ISPASS*, 2015.

[54] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *IEEE IC2E*, 2015.

[55] "Containers Not Virtual Machines Are the Future Cloud," http://www.linuxjournal.com/content/containers%E2%80%94not-virtual-machines%E2%80%94are-future-cloud.

[56] B.-C. Tak, C. Isci, S. S. Duri, N. Bila, S. Nadgowda, and J. Doran, "Understanding security implications of using containers in the cloud." in *USENIX annual technical conference*, 2017, pp. 313–319.

[57] O. Tunde-Onadele, J. He, T. Dai, and X. Gu, "A study on container vulnerability exploit detection," in *2019 ieee international conference on cloud engineering (IC2E)*. IEEE, 2019, pp. 121–127.

[58] singularityhub, "stools: singularity container tools for continuous integration and quality assessment." [Online]. Available: https://github.com/singularityhub/stools

[59] NIST, "National Vulnerability Database (NVD)." [Online]. Available: https://nvd.nist.gov/

[60] "Using the Dirty Pipe Vulnerability to Break Out from Containers," https://www.datadoghq.com/blog/engineering/dirty-pipe-container-escape-poc/.

[61] "CVE-2022-23648: Kubernetes Container Escape Using Containerd CRI Plugin and Mitigation," https://www.crowdstrike.com/blog/understanding-cve-2022-23648-kubernetes-vulnerability/.

[62] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "ContainerLeaks: Emerging security threats of information leakages in container clouds," in *IEEE/IFIP DSN*, 2017.

[63] X. Gao, G. Liu, Z. Xu, H. Wang, L. Li, and X. Wang, "Investigating security vulnerabilities in a hot data center with reduced cooling redundancy," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[64] "gVisor," https://github.com/google/gvisor.

[65] "About Kata Containers," https://katacontainers.io/.

[66] G. Myles and C. Collberg, "K-gram based software birthmarks," in *Proceedings of the 2005 ACM symposium on Applied computing*. ACM, 2005, pp. 314–318.

[67] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *Journal in Computer Virology*, vol. 1, no. 1-2, pp. 13–23, 2005.

[68] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (english version)."

[69] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *International Conference on Information and Communications Security*. Springer, 2008, pp. 238–255.

[70] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. ACM, 2013, p. 4.

[71] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 329–338.

[72] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 406–415.

[73] Y. David and E. Yahav, "Tracelet-based code search in executables," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 349–360, 2014.

[74] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 709–724.

[75] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of ACM CCS*, 2016.

[76] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code," in *Proceedings of NDSS*, 2016.

[77] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of ACM CCS*, 2017.

[78] F. Zuo, X. Li, Z. Zhang, P. Young, L. Luo, and Q. Zeng, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *NDSS*, 2019.

[79] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[80] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: cross-version binary code similarity detection with dnn," in *Proceedings of ASE*, 2018.

[81] Y. Ji, L. Cui, and H. H. Huang, "BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network," in *16th ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*, 2021.

[82] "The llvm compiler infrastructure," https://llvm.org/.

[83] "Valgrind Vex IR," last accessed 2023. [Online]. Available: https://valgrind.org/docs/manual/writing-tools.html

[84] J. Kessenich, B. Ouriel, and R. Krisch, "Spir-v specification," *Khronos Group*, vol. 3, p. 17, 2018.

[85] "Multi-Level Intermediate Representation," last accessed 2023. [Online]. Available: https://mlir.llvm.org

[86] "py2llvm," https://github.com/jdavid/py2llvm.

[87] "LLVM compiler for python based on py2llvm," https://github.com/aherlihy/PythonLLVM.

[88] "Generate llvm ir from matlab source code," https://github.com/leonardoaraujosantos/Matlab_LLVM_Frontend.

[89] "Retdec is a retargetable machine-code decompiler based on llvm." https://github.com/avast/retdec.

[90] "Framework for lifting x86, amd64, aarch64, sparc32, and sparc64 program binaries to llvm bitcode." https://github.com/lifting-bits/mcsema.

[91] "This tool statically (aot) translates (or raises) binaries to llvm ir." https://github.com/Microsoft/llvm-mctoll.

[92] X. Ren, M. Ho, J. Ming, Y. Lei, and L. Li, "Unleashing the hidden power of compiler optimization on binary code difference: An empirical study," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 142–157.

[93] "Llvm 8.0.0 release notes," https://releases.llvm.org/8.0.0/docs/ReleaseNotes.html.

[94] Y. Zhang, D. Shen, G. Wang, Z. Gan, R. Henao, and L. Carin, "Deconvolutional paragraph representation learning," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[95] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*. PMLR, 2014, pp. 1188–1196.

[96] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[97] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[98] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[99] Y. Ji, L. Cui, and H. H. Huang, "Vestige: Identifying Binary Code Provenance for Vulnerability Detection," in *International Conference on Applied Cryptography and Network Security (ACNS)*. Springer, 2021, pp. 287–310.

[100] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[101] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *International Conference on Machine Learning*, 2016.

[102] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[103] B. Bowman, C. Laprade, Y. Ji, and H. H. Huang, "Detecting Lateral Movement in Enterprise Computer Networks with Unsupervised Graph AI," in *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[104] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 815–823.

[105] D. E. Knuth, *The art of computer programming*. Pearson Education, 1997, vol. 3.

[106] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, pp. 604–613.

[107] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Communications of the ACM*, vol. 51, no. 1, pp. 117–122, 2008.

[108] F. Aurenhammer, "Voronoi diagrams—a survey of a fundamental geometric data structure," *ACM Computing Surveys (CSUR)*, vol. 23, no. 3, pp. 345–405, 1991.

[109] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 3, pp. 209–226, 1977.

[110] S. Suri, "Locality Sensitive Hashing," 2019. [Online]. Available: https://sites.cs.ucsb.edu/~suri/cs235/LSH.pdf

[111] "CVE - CVE," https://cve.mitre.org/.

[112] "runC," https://github.com/opencontainers/runc.

[113] L. Liu, H. Wang, A. Wang, M. Xiao, Y. Cheng, and S. Chen, "Mind the gap: Broken promises of cpu reservations in containerized multi-tenant clouds," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 243–257.

[114] H. Huang, J. Rao, S. Wu, H. Jin, H. Jiang, H. Che, and X. Wu, "Towards exploiting cpu elasticity via efficient thread oversubscription," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, 2021, pp. 215–226.

[115] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella, "Iron: Isolating network-based cpu in container environments," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 313–328.

[116] X. Gao, J. Xiao, H. Wang, and A. Stavrou, "Understanding the security implication of aborting live migration," *IEEE Transactions on Cloud Computing*, 2020.

[117] J. Connelly, T. Roberts, X. Gao, J. Xiao, H. Wang, and A. Stavrou, "CloudSkulk: A Nested Virtual Machine Based Rootkit and Its Detection," in *The IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021.

[118] K. Huang, X. Gao, F. Zhang, and J. Xiao, "Coms: customer oriented migration service," in *2017 IEEE 10th international conference on cloud computing (CLOUD)*. IEEE, 2017, pp. 692–695.

[119] Lei Lu, Xing Gao, and Jidong Xiao, "TwinHype: A Novel Approach to Reduce Cloud Downtime," in *The Annual Network and Distributed System Security Symposium (NDSS), Poster Session*, 2016.

[120] X. Gao, Z. Gu, M. Kayaalp, and D. Pendarakis, "Kernel-based power consumption and isolation and defense against emerging power attacks," Jan. 3 2019, uS Patent App. 15/638,694.

[121] C. Imes, S. Hofmeyr, D. I. D. Kang, and J. P. Walters, "A case study and characterization of a many-socket, multi-tier numa hpc platform," in *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. IEEE, 2020, pp. 74–84.

[122] "What Is Hyper-Threading?" [Online]. Available: https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html

[123] "Kernel Virtual Machine," https://www.linux-kvm.org/page/Main_Page.

[124] "QEMU A generic and open source machine emulator and virtualizer," https://www.qemu.org/.

[125] "rust-vmm," https://github.com/rust-vmm.

[126] "virtiofs." [Online]. Available: https://virtio-fs.gitlab.io/

[127] "Disk I/O Performance of Kata Containers." [Online]. Available: https://www.stackhpc.com/images/IO-Performance-of-Kata-Containers-TheNewStack.pdf

[128] "libkrun." [Online]. Available: https://github.com/containers/libkrun

[129] "virtio-vsock." [Online]. Available: https://chromium.googlesource.com/chromiumos/platform2/+/9e91613d2da1b3d6cfb1c77681444e688ce99cf4/vm_tools/docs/vsock.md

[130] "vsock(7) — Linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man7/vsock.7.html

[131] NVIDIA, "NVIDIA Multi-Instance GPU and NVIDIA Virtual Compute Server."

[132] ——, "NVIDIA Virtual GPU Software Documentation."

[133] ——, "NVIDIA Multi-Instance GPU User Guide."

[134] J. Thalheim, P. Okelmann, H. Unnibhavi, R. Gouicem, and P. Bhatotia, "Vmsh: hypervisor-agnostic guest overlays for vms," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 678–696.

[135] "Lonestar6 - texas advanced computing center," https://www.tacc.utexas.edu/systems/lonestar6.

[136] "Jetstream2: Indiana university," https://jetstream-cloud.org/.

[137] "Chameleon: A configurable experimental environment for large-scale edge to cloud research," https://chameleoncloud.org/.

[138] Y. Gui, Y. Wan, H. Zhang, H. Huang, Y. Sui, G. Xu, Z. Shao, and H. Jin, "Cross-language binary-source code matching with intermediate representations," *arXiv preprint arXiv:2201.07420*, 2022.

[139] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Thirtieth AAAI conference on artificial intelligence*, 2016.

[140] G. Mathew and K. T. Stolee, "Cross-language code search using static and dynamic analyses," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 205–217.

[141] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2015, pp. 131–140.

[142] "UNT-DEEN | College of Engineering," https://engineering.unt.edu/deen.

[143] L. Wang, A. Malladi, and Y. Ji, "Efficient sparse deep neural network computation on gpu (poster)," *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2022.

[144] "Csta," https://dfwcsta.com/page/texas-csta-chapters-conference-2023, 2023.

[145] "Digital divas 2023," https://digital-divas.weebly.com/.

Data Management Plan

This data management plan covers the data to be generated by the proposed project in the period of three years. The data of this research will be acquired, preserved, and shared, where appropriate, in accordance with the NSF policy and guidance regarding data sharing and stewardship. Our data management and sharing practice will also follow the policies of the PIs' institutions on intellectual property, privacy, security, and record retention and management, and will respect publication copyright and confidentiality.

Standards of data and metadata. The proposed research will mainly include observational data, collected code corpus, trained natural language processing (NLP) model, trained graph neural network (GNN) model, results for evaluation, as well as reports and documents pertaining to outreach and educational activities. Research data will be saved in text files, Word, Portable Document Format (PDF), Rich Text Format (RTF), Python Format (PY), Checkpoint File Format (CKPT), and Excel (XLS) format. Figures will be produced in JPEG and TIFF formats. We will organize each type of data into a folder and offer a README file for every folder that describes how the data are formatted and the programs are executed. We will also provide information on the code's contributors and license for sharing.

Policies for data access and sharing. Data will be made accessible to the research community immediately upon publication. Issues related to intellectual property, privacy, and confidentiality are not expected with the data generated by this project. Both PIs will maintain control of all project data and be in control of access to the data stored at UNT and UD. When publishing our source code and data, we will use one of the following Free Software Foundation licenses:

- GNU General Public License: http://www.gnu.org/copyleft/gpl.html
- GNU Lesser General Public License: http://www.gnu.org/licenses/lgpl.html

Policies and provisions for data storage, re-use, and re-distribution All project data (including any relevant data not published) will be maintained electronically at each PI's institute. Both PI Ji and PI Gao have already bought their own servers. All the datasets will be stored on the servers. Moreover, we will use RAID for robust data storage on hard disk drivers. PI Ji will set up a git repository to share and synchronize datasets with PI Gao. Any data obtained will be retained for at least 5 years after the end of this project. Any outreach and programs that make use of these data will be documented and any presentations, flyers, and supplemental information will be archived as part of the data storage procedures for this project. These materials will be retained for at least 5 years after the event has taken place.