Jun Duan, Kevin W. Hamlen, and Benjamin Ferrell. “Better Late Than Never: An *n*-Variant Framework of Verification for Java Source Code on CPU×GPU Hybrid Platform.” In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2019.

Better Late Than Never: An *n*-Variant Framework of Verification for Java Source Code on CPU × GPU Hybrid Platform

Jun Duan
jun.duan@utdallas.edu
The University of Texas at Dallas

Kevin W. Hamlen
hamlen@utdallas.edu
The University of Texas at Dallas

Benjamin Ferrell
benjamin.ferrell@utdallas.edu
The University of Texas at Dallas

ABSTRACT

A method of detecting malicious intrusions and runtime faults in software is proposed, which replicates untrusted computations onto two diverse but often co-located instruction architectures: CPU and GPU. Divergence between the replicated computations signals an intrusion or fault, such as a zero-day exploit. A prototype implementation for Java demonstrates that the approach is realizable in practice, and can successfully detect exploitation of Java VM and runtime system vulnerabilities even when the vulnerabilities are not known in advance to defenders.

To achieve acceptable performance, it is shown that GPU parallelism can be leveraged to rapidly validate CPU computations that would otherwise exhibit unacceptable performance if executed on GPU alone. The resulting system detects anomalies in CPU computations on a short delay, during which the GPU replica quickly validates many CPU computation fragments in parallel in order to catch up with the CPU computation. Significant differences between the CPU and GPU computational models lead to high natural diversity between the replicas, affording detection of large exploit classes without laborious manual diversification of the code.

CCS CONCEPTS

• **Security and privacy → Logic and verification**; **Software security engineering**; • **Software and its engineering**;

KEYWORDS

n-variant; Java; software reliability; intrusion detection; software exploit detection; software engineering

ACM Reference Format:
Jun Duan, Kevin W. Hamlen, and Benjamin Ferrell. 2019. Better Late Than Never: An *n*-Variant Framework of Verification for Java Source Code on CPU × GPU Hybrid Platform. In *The 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC'19), June 22–29, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307681.3326604

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HPDC '19, June 22–29, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-6670-0/19/06…$15.00
https://doi.org/10.1145/3307681.3326604

1 INTRODUCTION

N-variant systems [13] detect intrusions and other runtime anomalies in software by deploying diverse replicas of the software and monitoring their parallel execution for computational divergence. Divergence between the computations indicates that one or more replicas have exercised functionalities that were unintended by the program's developers, and that were therefore not replicated consistently across all the copies. The *n*-variant approach has been used for detecting memory corruption vulnerabilities in C/C++ programs [54], monitoring user-space processes [44], defending data corruption attacks [38], and securing embedded systems [3].

Unfortunately, one major barrier to the realization of effective *n*-variant systems in practice has been the high difficulty and cost associated with creating and maintaining software copies that are appropriately diverse (not replicating bugs or vulnerabilities), yet consistent (preserving all desired program features). Achieving this can entail employing multiple independent software development teams, which can potentially multiply the cost and time associated with the project by a factor of *n* [5].

This high cost of independent, manual cultivation of diversity has led to a search for automated software diversity. For example, compilers have been proposed as natural diversity-introduction vehicles [27], since they enjoy a range of options when translating source programs to distributable object code, including various possible object code and process memory layouts. However, many large classes of software attacks exploit low-level details that are fundamental to the target hardware architecture, and that are therefore difficult for compilers to meaningfully diversify. For example, Address Space Layout Randomization (ASLR) defenses, which randomize section base addresses in process memory at load-time, have proven vulnerable to derandomization attacks [47] that exploit the prevalence of relative-address instruction operands in CISC instruction sets to learn the randomized addresses. Similarly, return-oriented programming [46] and counterfeit object-oriented programming attacks [45] abuse the semantics of return and call instructions, which is difficult to avoid when compiling to architectures with those instruction semantics.

Our research in this paper is inspired by the observation that modern computing systems increasingly have two very different yet powerful instruction architectures available to them: CPU and GPU. This potential source of computational diversity has gone relatively unutilized as an opportunity to detect malicious software intrusions through *n*-variant computation. To explore this opportunity, we introduce Java Gpu-Assisted *N*-variant Guardian

(J-GANG), a system that replicates Java computations onto CPU-GPU hybrid architectures and runs them concurrently in order to detect divergence-causing intrusions.

GPU computing models differ substantially from typical CPU computing models. This diversity offers many attractive opportunities for robust intrusion detection, but is also a source of significant technical challenges. In general, GPU architectures suffer poor performance on computations with few threads; their advantages are only seen on computations with hundreds or thousands of simple but independent workloads. However, most Java computations offer only limited parallelism on the order of a few threads. Running Java computations in a brute-force fashion on GPUs therefore risks bottlenecking the system; the GPU variant could lag hopelessly behind the CPU variant, forcing the latter to wait.

Our approach therefore instead adopts an asynchronized model in which the CPU variant runs at full speed, logging its results at selected program *checkpoints* in the form of JVM state snapshots. A sequence $(\sigma_0, \ldots, \sigma_k)$ of such snapshots can be replicated and validated by a GPU using k concurrent workers, each of which validates the (σ_i, σ_{i+1}) portion of the computation by starting at state σ_i as a pre-condition and confirming that it reaches state σ_{i+1} as a post-condition ($\forall i \in 0..k-1$). The computation is correct only if all these fragments pass validation. This allows the GPU to catch up to the CPU computation in spurts—the more it lags behind, the more opportunity for parallelism arises, since it can greedily consume more snapshots and validate them concurrently.

The high dissimilarity between GPU and CPU models of Java computation state allow J-GANG to detect many important vulnerability classes. For example, attacks that exploit memory corruption vulnerabilities to hijack return addresses on the stack have a different effect upon GPU computations, since the GPU model has no explicit call stack with in-memory return addresses to corrupt. Moreover, our detection approach conservatively assumes that all exploited vulnerabilities are unknown to defenders (zero-days). No explicit knowledge of vulnerabilities is used to avoid preserving them in the GPU replica; divergences arise purely from the natural dissimilarity between the two instruction architectures.

The contributions of J-GANG can be summarized as follows:

- We introduce the first n-variant system for Java computation verification based on architectural differences between GPU and CPU.
- To harmonize the dissimilar performance advantages of the two architectures, we introduce an asynchronous *trust-but-verify n*-variant model, in which single- or few-threaded CPU computations are validated by many-threaded GPU computations on a short delay. This allows the GPU computation to quickly verify many iterations of CPU-executed loops concurrently.
- A prototype implementation establishes rules of translation from Java source code on the host side into GPU-executable kernel code, which offers a possible solution to facilitate GPU execution of general Java source code in future work.
- Evaluation of J-GANG on exploits of eight real-world JVM vulnerabilities exhibits reliable detection at reasonable overheads, even when the vulnerabilities are treated as zero-days (no vulnerability-specific defenses introduced).

Listing 1: Exploit of JDK-5091921 (JavaSE 1.6, x86/x64 Win7)

```
1  int i = 0;
2  int j = Integer.MAX_VALUE;
3  boolean test = false;
4  while (i >= 0) {
5    i++;
6    if(i > j) {
7      test = true;
8       break;
9    }
10 }
11 System.out.println("Value of i: " + i);
12 if(test) i = 1;
13 System.out.println("Value of i: " + i);
```

Listing 2: Exploit of JDK-8189172 (JavaSE 1.8, x86/x64 Win7)

```
1  double b = 1.0 / 3.0;
2  double e = 2.0;
3  double r = Math.pow(b, e);
4  double n = b * b;
5  while (r == n) {
6       b += 1.0 / 3.0;
7       n = Math.pow(b, e);
8       r = b * b; }
9  println("b=" + b + " n=" + n + " r=" + r);
```

- Methods of tracing variables and local data analysis are extensively tested, and we study the connection between overhead and state snapshot logs. Based on the relation, we control overhead for suitable variable-tracking jobs.

The remainder of the paper is arranged as follows. Section 2 describes the system design and defines correctness. Section 3 details our prototype implementation. Experimental methodology and evaluation are discussed in Section 4. Section 5 introduces related works. Finally, Section 6 concludes.

2 SYSTEM DESIGN

2.1 Divergence Between Executions

Listing 1 exhibits a JVM vulnerability related to around 30 bugs and numerous DoS attacks against Java SE 1.6, and that was later identified as a root cause of array overflows, server VM crashes, and a variety of other potential software compromises before it was patched.[1] It returns different unstable values of i on each execution, and also prints the false result, *"Value of i: 1"* in line 13 when it should report overflowed value −2147483648. The flaw is an incorrect optimization in the (CPU-based) HotSpot compiler, which breaks integer overflow detection in certain loops. However, running this code in our J-GANG system as a GPU computation yields correct results, because GPUs apply a very different procedure for optimizing the loop. This natural difference in behavior offers a potential opportunity to detect the error without advance knowledge of the bug.

[1] https://bugs.java.com/view_bug.do?bug_id=5091921

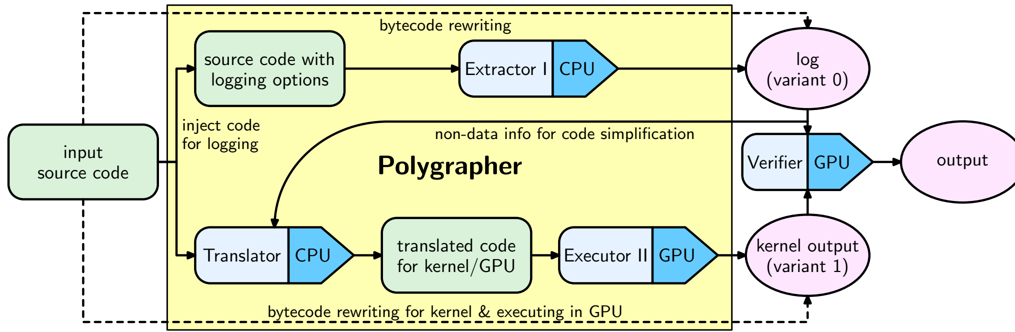


Figure 1: J-GANG Architecture

Listing 2 likewise demonstrates an exploit of JVM bug JDK-8189172,[2] which embodies an imprecision of floating point computations that in this case causes expressions $n \times n$ and n^2 to return unequal results. A correct JVM should loop infinitely, but unpatched JVMs halt with output $b = 4.\overline{9}$, $n = 24.\overline{9}$, $r = 24.999999999999993$. However, compiling the same code to a GPU architecture results in correct behavior—self-product and square yield equal results, and the program loops infinitely. Detecting this divergence of behavior has the potential to detect the exploit without the need to craft and deploy vulnerability-specific mitigations whose formulation require advance knowledge of the bug.

J-GANG detects both exploits by compiling the Java source code to two binary executables: (1) logging-enhanced Java bytecode, and (2) verification-enhanced OpenCL GPU code. The Java bytecode variant logs local state (e.g., variables b, e, r, and n in Listing 2) at the start of each loop iteration (line 5) and at loop exit (line 9). The GPU variant consumes this log stream in a verification loop. When consuming k available checkpoints, it spawns $k-1$ workers that each initialize their local variable states in accordance with different checkpoints σ_i ($i < k-1$). They then all execute one iteration of the loop in parallel, and confirm that the resulting states matches the succeeding checkpoints σ_{i+1}. The divergence is detected when the final worker obtains a different state than the CPU (e.g., equal values for r and n in Listing 2).

While both divergences could theoretically be detected by replicating programs to multiple, dissimilar CPU-based JVMs wherein at least one JVM emulates a GPGPU-style computational model, in practice there are at least two significant problems with this CPU-only approach. First, CPU-based emulation of GPU-style parallelism is highly inefficient. A CPU-based JVM that emulates the computational diversity of a GPGPU computation therefore cannot keep pace with the CPU computation it is seeking to verify, resulting in unacceptable performance bottlenecks.

Second, building and maintaining a new, dissimilar, production-level JVM is difficult and expensive, as witnessed by the fairly small and homogeneous set of production JVMs currently available despite over 25 years of Java infrastructure development. These JVMs intentionally offer little diversity, since diversity introduces maintainability and cross-compatibility issues. For example, the flaw demonstrated by Listing 2 has been reported across several JVMs by many users, probably because it is rooted in runtime library code shared by many CPU-based JVM implementations. Leveraging a CPU×CPU hybrid model potentially offers greater diversity by extending dissimilarities down to the hardware level, yet avoiding overheads suffered by network communications between machines.

[2] https://bugs.java.com/view_bug.do?bug_id=8189172

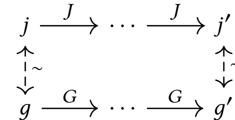


Figure 2: Semantic transparency

2.2 Model & TCB

Figure 1 shows the system architecture of J-GANG. The hardware differences between the two execution paths forms an ideal polygrapher, which is defined as a distributer to feed the executors with input. To generate the acceptable parallel equivalent states for CPU and GPU respectively, there are two working paths in the polygrapher. Since the input is Java source code, one path processes the original CPU execution. The other consists of an translation action and several processing behaviors of corresponding states expressed in the GPU. The two state streams are compared for semantic equality in an on-demand fashion.

The correctness of a compiler that transforms a source program (e.g., Java) into an object code program (e.g., Java bytecode or GPU bytecode) is defined in the literature in terms of *semantic transparency* (cf., [33]), which asserts that the source code semantics and the compiled object code semantics yield equivalent program states. In the case of two compilers (source-to-JVM and source-to-GPU), we therefore transitively define relation $\sim \in J \times G$ to be the equivalence relation between the two object languages—JVM states $j \in J$ and GPU states $g \in G$—that is preserved by the two compilers' semantic transparencies. Specifically, we define $\mathcal{JVM} \in (J, \rightarrow_J)$ to be a transition system that encodes the operational semantics of the Java bytecode virtual machine, such as ClassicJava [16] or Featherweight Java [26]. Similarly, define $GPU \in (G, \rightarrow_G)$ to be a transition system that encodes the operational semantics of GPU bytecode programs, such as PTX [21].

Figure 2 shows a commutative diagram illustrating how non-malicious executions that stay within the intended semantics of the two transition systems preserve relation $\sim$. As indicated by the diagram, this semantic equivalence is not necessarily step-wise; state equivalence is only checked periodically at checkpoints. This is important not only for performance, but also for reflecting differences in granularity between the two architectures. For example,

certain computational steps by the CPU execution engine might correspond to a series of multiple computational steps on a GPU.

All non-determinism sources (e.g., random number generation, scheduling, user input, clock checks) are treated as inputs by J-Gang and logged by the CPU variant as local state. In general, this leaves three scenarios that can potentially falsify transparency:

(1) one or both transition systems reach stuck states,
(2) one system reaches a final state before the other, or
(3) relation ~ is falsified.

Condition 1 corresponds to a failure of J-Gang's implementation (the compilers, runtime systems, or validator). For example, Java language features unsupported by the prototype implementation (see §3) yield stuck states. Condition 2 corresponds to premature termination, as exhibited by the example in Listing 2. The most significant form of falsification arises from condition 3, which corresponds to developer-unintended behaviors that differ between the two transition systems. These include memory corruption, arithmetic errors, and type confusions indicative of many Java exploits.

2.3 GPU Feature Limitations

Current GPU instruction architectures support only a small subset of operations available on CPUs. For example, reference types (objects) and methods are not directly expressible in the kernel part of programs parsed in either the CUDA or OpenCL platforms. Likewise, GPU kernel code cannot directly access main memory during computations, since to access the main memory by shared virtual memory (SVM) is an optional feature of OpenCL and still not perfectly supported by AMD in Windows. J-Gang therefore does not rely upon it.

While these limitations may initially seem prohibitive to our goal of replicating general JVM computations to GPUs, they actually serve to enhance J-Gang's ability to detect attacks within our asynchronized validation model. Any CPU operation that cannot be supported on GPU is idealized during source-to-GPU compilation as an opaque input-output relation defined by the CPU variant's computation. For example, objects are reduced to their integer hash codes on the GPU side, and calls to their methods become checkpoints whose local states include numeric indexes of the called method and the return site. This allows the GPU variant to verify that the same object and method is called. A separate worker then validates the callee's computation and its return, avoiding an explicit method call or call stack on the GPU side. Usually these caller and callee computations are validated concurrently by the GPU.

The idealization and opacity of these operations on the GPU side is a source of many opportunities for detection of malicious computations. For example, exploits that corrupt the JVM's call stack or method tables to hijack code control-flows almost never have the same effect on J-Gang's GPU computations, which have no explicit call stack or method tables, and that exercise independent, parallel workers instead of performing serial method calls.

2.4 Validation Modes

J-Gang can be configured to execute in two possible modes:

Static Mode. The CPU variant can be configured to execute to completion before delivering its checkpoint log, whereupon the GPU variant validates the entire computation. This mode can be useful for terminating computations that demand high realtime efficiency, and that do not require immediate validation.

Dynamic Mode. In this mode, the CPU variant streams its checkpoint log to the GPU variant as the computation progresses. The GPU variant consumes the stream opportunistically, discarding the consumed checkpoints. This is the preferred mode, since it reduces space overheads for checkpointing, accommodates non-terminating computations, and detects intrusions live.

2.5 Checkpointing

Local State. Checkpoints produced by the CPU variant consist of local variable values, heap values (e.g., object hash codes and fields), and a numeric token that uniquely identifies the current code point. To control overhead, only the subset of local variable and heap values that are accessed by the CPU variant between this checkpoint and the next are included in each checkpoint. While purely static liveness analysis of Java code can be challenging [39], we avoid many of these complexities by simply logging the variable values that are actually read and written by the CPU variant as it runs, and by placing checkpoints at significant meets and joins in the control-flow graph (e.g., function and loop entry and exit points). In this way we avoid the need to accurately compute heap liveness or reachability, and all static analyses are intraprocedural. Liveness and reachability approximations are only used as optimizations to avoid unnecessary checkpoints.

If the GPU variant attempts to access a state element that was not included in its source checkpoint, or modifies a state element not included in its destination checkpoint, it signals a divergence. Thus, checkpoints and any analyses used to generate them remain untrusted by the verifier.

Frame State. The GPU variant also maintains a *frame state* comprising portions of the heap that were introduced by previous (now discarded) checkpoints, and that remain reachable, but that do not appear in the current checkpoints undergoing validation. This reduces checkpoint sizes by providing a means to validate the values of variables that are not read or modified for large portions of the program, but that remain live. It is maintained outside the GPU kernel code, and consists of an idealized JVM state representation in which objects are expressed as hash codes and their fields are expressed as hash tables.

For example, variable e in Listing 2 remains live throughout the loop, but is only accessed in line 7. By including e in the frame state, we can omit e from checkpoints for computation fragments that do not concern e. Checkpoints that assume $e = 2.0$ as a precondition can nevertheless be validated by consulting the frame state. Like other variables, modifications of frame elements are reported in checkpoints, and are therefore validated by the GPU, resulting in changes to its frame state.

2.6 Translation

Translation of Java source code to J-Gang's hybrid architecture is summarized in Figure 3. For simplicity of presentation, we here represent Java source code as a core language consisting of variable assignments $v \leftarrow e$, method calls $v \leftarrow o.m(\vec{e})$ (which have been factored out of expressions into separate statements), sequences, loops, n-way branches, and exception-handlers. Translation function T

$$
\begin{aligned}
s &::= v \leftarrow e \mid v \leftarrow o.m(\vec{e}) \mid s_1;\ s_2 \mid \mathsf{loop}(e)\ s && \text{(statements)}\\
&\quad\mid \mathsf{branch}(e)\ s_1 \cdots s_n \mid \mathsf{try}\ s_1\ \mathsf{with}\ e \Rightarrow s_2 &&\\
e & && \text{(expressions)}\\
v & && \text{(variables)}\\
☑ & && \text{(checkpoints)}
\end{aligned}
$$

$$
\begin{aligned}
T(v \leftarrow o.m(\vec{e})) &= (\vec{v}_{tmp} \leftarrow \vec{e};\ ☑;\ v \leftarrow \mathsf{call}(o.m, \vec{v}_{tmp});\ ☑)\\
T(s_1;\ s_2) &= T(s_1);\ T(s_2)\\
T(\mathsf{loop}(e)\ s) &= v_{tmp} \leftarrow e;\ ☑;\ \mathsf{loop}(v_{tmp})\ (T(s);\ v_{tmp} \leftarrow e;\ ☑)\\
T(\mathsf{branch}(e)\ s_1 \cdots s_n) &= v_{tmp} \leftarrow e;\ ☑;\ \mathsf{branch}(v_{tmp})\ T(s_1) \cdots T(s_n)\\
T(\mathsf{try}\ s_1\ \mathsf{with}\ e \Rightarrow s_2) &= \mathsf{try}\ T(s_1)\ \mathsf{with}\ e \Rightarrow (☑;\ T(s_2))
\end{aligned}
$$

Figure 3: Translation of Java source to CPU×GPU code

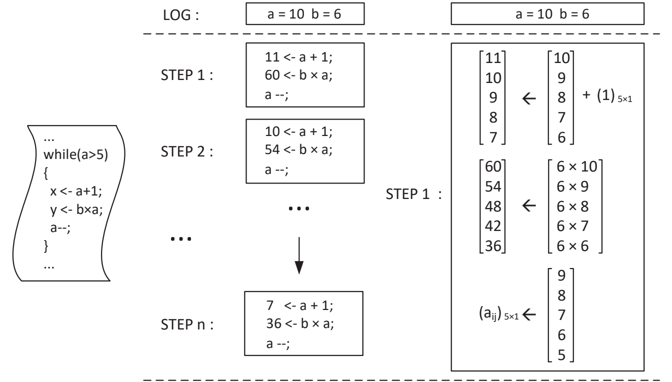


Figure 4: J-Gang computations for CPU (left) vs. GPU (right)

maps these programs to instrumented source code programs that can be compiled to native CPU/GPU architectures.

The translation process adds checkpoint operations ☑, which have a different operational semantics depending on the target architecture. In the CPU variant, checkpoints log the local state to the verification log. In the GPU variant, checkpoints read the log to initialize the local state at the start of a worker computation, and to validate the local state at the end of each worker computation. Translation of loops, branches, and exception handlers entails adding checkpoints to meets and joins in the program's control-flow graph. To check loop and branch conditions, they are assigned to translator-introduced temporary variables v_{tmp}, which contribute to the local state and hence undergo checkpointing.

Translation of method calls invokes a call verification handler $\mathsf{call}(o.m, \vec{v})$ whose semantics likewise differ between the two architectures. On CPUs, object o's hashcode is logged to the checkpoint, method m of object o is called with arguments $\vec{v}$, and its return value is logged on return. On GPUs, where explicit calls do not exist, the logged hashcode is verified to equal the GPU state's object argument, and the return value is simply retrieved from the log file and used as the result. This works because the checkpointing placement ensures that a separate GPU worker always verifies the correctness of this return value when validating the callee's computation. (If the callee is not Java code, as in the case of JVM runtime system calls, this treatment simulates the GPU calling the external library with the same arguments and receiving the same result value.)

Each checkpoint also logs a program label that uniquely identifies the location of the checkpoint in the code. Thus, the GPU code consists entirely of a single function beginning with a branch that consults this label to conditionally jump to the code fragment being checked. Each GPU worker thereby executes a code fragment that begins at one checkpoint and ends at the next, and that consists entirely of side effect-free computational expressions suitable for GPU kernel code.

Figure 4 depicts the resulting execution streams for a simple loop. The CPU variant (left) executes the loop body iteratively in a serial stream, outputting one checkpoint for each iteration (and one additional one at start). A GPU variant (right) with n workers consumes all available checkpoint-pairs simultaneously, simulatig all iterations of the loop in parallel to validate the computation.

2.7 Verification Time Complexity

Modern GPGPU architectures are most efficient when threads execute *homogeneously*—i.e., each group of k threads executes the *same code* in lock-step (on possibly different data), and there is no significant communication between threads in the group. For example, Nvidia's CUDA GPGPU architecture supports a Same Instruction Multiple Data (SIMD) model (as well as less efficient but more flexible MIMD models) [34]. On a GPGPU architecture with a single thread group of size k, J-Gang's GPU variant can obey this homogeneity constraint to achieve high efficiency, and thus keep pace with the CPU variant even after lagging behind the CPU by a factor of k, as shown by the following theorem.

Theorem. *If the time complexity of the CPU code is $O(f(n))$, then the time complexity of the GPU-translated code on an architecture with k homogenous threads is $O(f(n)/k)$.*

Proof Sketch. Code size c is constant relative to the input size n. By pigeon-hole principle, a checkpoint sequence of length $O(f(n))$ must therefore contain $\Omega(f(n)/c) = \Omega(f(n))$ checkpoint pairs that span *identical* code fragments. Translation function T (see §2.6) executes these homogeneous fragments in blocks of k for a total runtime of $O(f(n)/k)$. □

In practice this means that even though each GPU thread's serial computing speed is less than that of a typical CPU, with reasonably large k the GPU variant nevertheless keeps pace with the CPU. This allows J-Gang to scale to long computations.

3 IMPLEMENTATION

To test and evaluate J-Gang, we implemented an extensive translation infrastructure from Java source to GPU kernel code. This includes a new Java package handler implementation, a CPU-GPU communication library for live data logging and retrieving, translation from host code to kernel code, and procedural automation.

Figure 5 depicts the procedure and interaction between source code and processing units in static mode. (Dynamic mode omits python scripts and reloads modified classes with class-loaders.) It shows how the GPU Kernel code of verification for Java Aparapi is generated from source code and how we record the status of variables and create checkpoints in the kernel in basic mode.

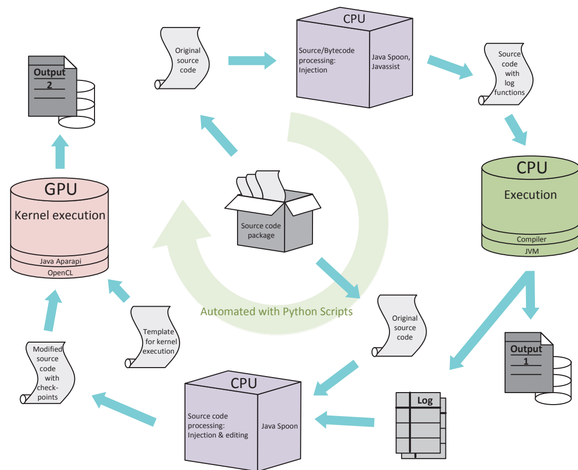


Figure 5: Procedure of variants generation and execution

Figure 6 lists four sample code fragments representing the elements of Figure 5. All code shown in the figure is generated automatically by J-Gang, including source code with log functions, and code instrumented with checkpoints. Reading and writing of log tables to align the data of loops for parallel verification (see Fig. 4) is also automated. This is shown in Log and Modified code elements of the figure. In addition, the code with check-points in the rightmost side represents the initial GPU kernel code from Java Spoon (before optimizations based on dynamic analysis are applied, such as granularity adjustment).

The project is implemented in Java and Python on a machine with Intel Xeon CPU @3.4GHz, 64GB memory, and an AMD FirePro W5100 Graphic Card. Supporting infrastructure includes Java Aparapi, Java Spoon [41], and Javassist [11]. Aparapi by AMD provides Java bindings to enable the host to call APIs from OpenCL; it is the portal for Java code execution on GPU, and masks OpenCL's more detailed operations and memory management in the devices. For example, it automatically switches from Java's multi-threading and GPU task scheduling, making its style of kernel code more accordant with Java code. Java Spoon by Inria is used as a language-parsing and code-injecting tool to log variable information and transform the input source. Javassist offers the ability to rewrite code from input at the bytecode level.

3.1 Source Language Limitations

Since our prototype is implemented atop Java Spoon and Java Aparapi, it is presently limited to Java code that can be parsed by those tools. Aparapi offers a Java-style grammar wrapper on the kernel code of OpenCL, which is based on the C99 standard and does not support Java-level multi-threading or certain higher-order OOP constructs (e.g., first-class lambdas). For exact limitations, please see the documentation of the aforementioned tools. Our prototype follows the basic Java SE standard, and therefore does not yet support language features new to subsequent Java versions. In addition, some language optimization will also be limited due to the current GPU and CPU's architecture of communication. For example, the optimization mentioned in Figure 4 for nested loops or recursive function will be impossible.

3.2 Bytecode Analysis

J-Gang uses bytecode analysis to log executions and dynamically rewrite GPU kernels (the dashed lines in Figure 1). In dynamic mode, the variables in the system are visited while executing the original source code. We wrote a toolkit package on Javassist for this task, since no convenient tool in the market currently provides functions for the Java language to visit local variables of methods in JVM at runtime. Java language extentions, such as AspectJ, offer indirect ways to achieve this, such as refactoring source code to expose local variables at compile time. Javassist and ASM offer manipulation in bytecode.

Figure 7 illustrates our procedure for logging local variable state. To locate the local variables in a Java bytecode method, the indices of local variables are first tracked by inspecting the opcode `iinc` and opcode family of `xload(_a)` and `xstore(_a)` of the method. From this we create the bytecode of the log statement with acquired line numbers and variables' indices of these opcodes, and in-line it into the method. Executing the instrumented method streams the log of variables to the verifier.

To dynamically rewrite the GPU kernel code, the source code is first translated to executable statements for the kernel and converted to its bytecode. This creates the code block that will be executed on the GPU. To make it executable, we compile an empty kernel template to bytecode and inject the bytecode of the code block. The newly generated kernel must be compiled and dynamically loaded before the compiling procedure for the whole source code starts. This is because the template kernel has already registered in the JVM before the generation of the new kernel. This one-time reloading initializes a nonstop procedure from input source code directly to execution in the GPU, achieving live, streaming computation validation.

3.3 Primitives & References

In the static mode, Java Spoon is used to parse the input source code. All the statements about initialization and assignment of variables are first located with their line numbers. In this stage, the update sites of variables are analyzed for liveness, and duplicately-named variables are assigned unique indexes in the log.

Java's primitive types are all recorded directly into logs, since the OpenCL kernel uses the same data types during verification. For example, values of type `char` are logged as `unsigned short`. To log reference types, a method of lightweight recording is chosen: The system tracks references' hashcodes to monitor their changes, since the GPU kernel lacks first-class references. All non-primitive objects or attributes can be disassembled or converted into primitives [20], affording verification of all references by the GPU.

3.4 State Consistency

Before verification starts, an initial memory state must be prepared so that both variants can begin computation in equivalent states. This *pre-state* corresponds to the precondition of a Hoare Triple. To keep the state size tractable, it is desirable to restrict each pre-state to only those variables that are referenced by the computation fragment being verified. To compose the pre-state, the code block to be verified and its line numbers are analyzed with Java Spoon so that relevant variables can be selected out. Each selected variable's

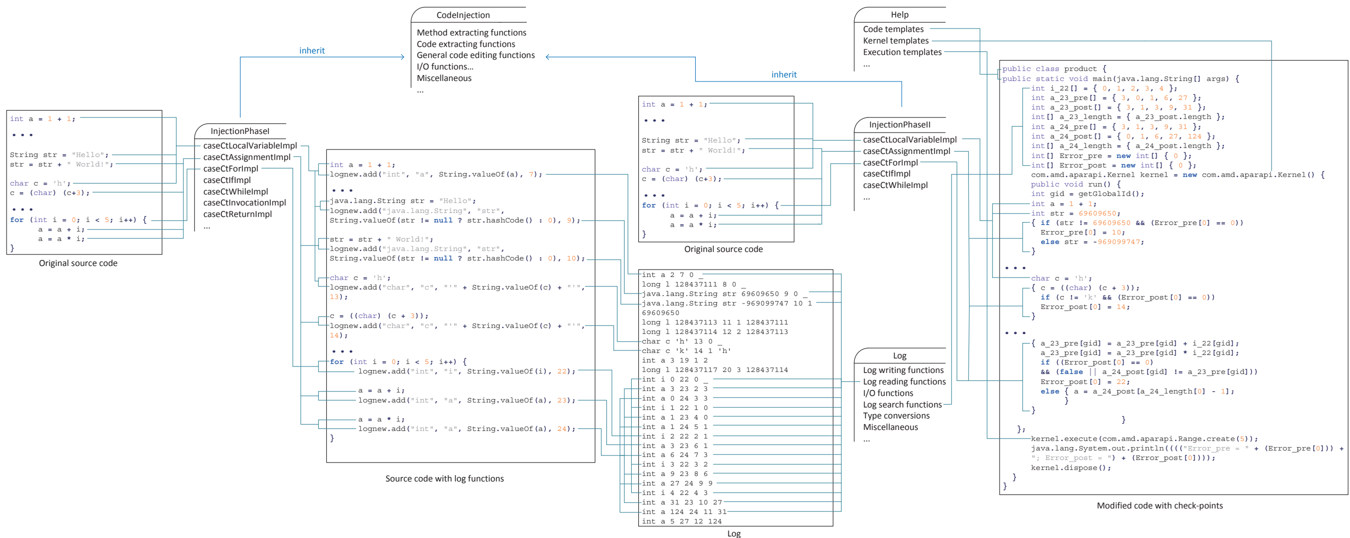


Figure 6: Code and checkpoints generation procedure

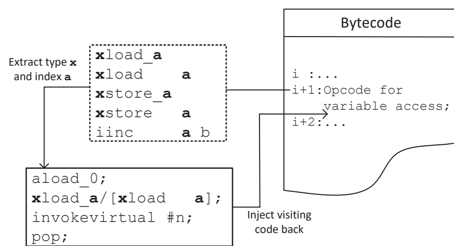


Figure 7: Bytecode injection to dynamically visit a variable

pre-state value is determined by identifying its last update in the checkpoint log, or its value in the frame state if the checkpoint log contains no updates.

To take advantage of parallel computing, J-Gang represents pre-state variables in different ways depending on the control-flow structures that contextualize each computational fragment being verified. Sequential and conditional control-flows offer only small opportunities for parallelism, so their variable values are stored separately in local memory. However, variables in (non-nested) loops are arranged into arrays and loaded into global memory for parallel verification. Our prototype does not yet perform this optimization for inner loops of nested loops, since doing so introduces complexities related to dynamically generating kernel code that anticipates how the various nesting levels interleave at runtime. This is an optimization we intend to pursue in future work.

3.5 GPU-based Verification

Executor II in Figure 1 is implemented as GPU kernel code analogous to the code that executes on the CPU. It first loads the initial computation state (pre-state) reported by the CPU version, and then compares the generated result state with the logged post-state. To leverage the performance strengths of GPGPU computing, J-Gang implements GPU code that enjoys two forms of parallelism: (1) Loops are parallelized into concurrent verification of their iterations, as shown in Figure 4. (2) Comparison of the post-state derived by the GPU to the one reported by the CPU is parallelized into concurrent comparisons of state partitions.

In the first part, when the system verifies the iterations of a loop in parallel, the values in each iteration for a variable in both the pre-state and post-state are aligned into an array, which affords efficient concurrent access by GPU kernel code. This asymptotically decreases the verification time by one layer of the loop, as proved in Section 2.7. With k parallel processing units in the GPU, the number of iterations is reduced from n to n/k, where n is the number of CPU loop iterations.

In the second part, we efficiently compare the two post-states from the executors by converting them into two byte arrays. Bitwise XOR computation can be performed on the arrays to efficiently compare them for equality. On the GPU, this XOR computation is parallelized among all available workers. This method significantly accelerates state comparisons, which are otherwise slow on serial architectures since the states can be large.

3.6 Code Pruning

When some part of the source code is never executed by Executor I (CPU) or has no effect upon the computation state (e.g., non-executed branches or effect-free code), this part can be trimmed from original code before the translation for Executor II (GPU). For example, it is not necessary to keep the discordant branches in the second execution since these are unreachable. This optimization is safe because divergent computations that include such code blocks are guaranteed to still exhibit divergence when omitting the effect-free blocks.

To implement this optimization, line numbers of variables are inspected in the log to determine the direction of flow before the translation. We record line numbers of variables with their updated values together during the checkpointing step. All statements, including those in branches, are tagged by the line numbers. During

the first execution, the values of variables in statements visited by program flow are logged. These recorded lines indicate which branch updated the variables. By checking the number, we deduce which branches can be omitted from verification. If there is no variable recorded in the branch, the segment of code can be trimmed since it is effect-free.

4 EVALUATION

Our experimental evaluation of J-Gang is grouped into vulnerability detection accuracy and runtime performance. The evaluation architecture is the same as the development framework reported in Section 3, and publicly available, independently authored Java input programs are selected from diverse sources for correctness and performance tests. Programs with different time complexity and utility are chosen to test running performance. Also, a group of relatively new Java bugs are selected from Oracle Java Bug Database to test the accuracy and utility of the framework for real-world scenarios. All bugs are treated as zero-days—no vulnerability-specific mitigations or controls are deployed for any of the experiments.

In the experiments, checkpointing is closely related to overhead. To control this trade-off, the granularity of checking can be flexibly tuned from the finest-grained level (checking every variable update immediately) to coarser-grained levels (checking variable updates after code blocks or function returns). For example, checkpoints can be inserted after each line of code within a loop, or only before and after the loop for greater efficiency. Coarser checking requires fewer checkpoints but potentially larger states to check at each checkpoint.

4.1 Running Efficiency

Performance evaluation of J-Gang can be characterized in terms of two metrics: (1) overall runtime overhead of the instrumented CPU computation, and (2) the delay between time-of-exploit and exploit-detection by the GPU verifier. Overheads measured by the first metric are primarily due to the extra time needed to log checkpoints for verification. Checkpointing is partly asynchronous, but there is still overhead incurred by initializing and spawning the asynchronous I/O. Overheads measured by the second metric are primarily driven by the size of the checkpoint stream, and are therefore measured in reciprocal-bandwidth (ns/B).

Our evaluations consider two categories of test application: classic algorithms (which afford investigation of time/space complexity effects, memory update frequency, and highly optimized code loops), and practical utilities (which examine applicability of J-Gang to real-world software products). The latter include website-downloaders, compression tools, and image editors. They are randomly chosen for the testing, and demonstrate our approach's generality and versatility. Selected programs are all from independent authors and were tested for correctness before evaluation. Each test data point reported is an average over hundreds of trials.

Performance is reported for both the static and the dynamic verification mode. In static mode, the CPU computation runs at full speed and produces a complete log of checkpoints, which is verified by the GPU after the computation completes. In dynamic mode, the checkpoint log is consumed opportunistically by the GPU verifier as the CPU computation progresses, affording live, parallel validation of the computation. The static mode therefore incurs lower I/O overheads, but has the disadvantage of building a larger checkpoint log and offering only retroactive detection of exploits. The dynamic mode incurs higher I/O costs but does not need to retain the full checkpoint log in memory or on disk, and detects exploits on a short delay.

Table 1 and Figure 8e report runtime overheads for the first category of tests (classic algorithms). Figures 8(a–d) report overheads for the second category (practical applcations). The results indicate that tight, numerically intensive computations incur high overheads (due to the high cost of frequent checkpointing relative to streamlined mathematical computations), but most of the practical applications perform well under J-Gang. For example, utility programs running in static mode show overhead ratios of less than $\leq 1.008\%$, and average overheads of less than 7% in dynamic mode. All overheads are under 10% except for the outlier in Figure 8d for unzip files, which is investigated in more detail below.

The experiments reported here do not include any manual granularity tuning; we allowed J-Gang to select checkpoint locations, frame state update frequency, and loop verification parallelizations purely automatically. To better support the short, computationally intensive algorithms in Table 1, we conjecture that a less frequent, time interval-based checkpointing regimen would perform better for such algorithms. Figure 8e investigates this conjecture by adjusting the checkpointing granularity for the binary search experiment, resulting in a more acceptable overhead of about 20%. Tuning the granularity in this way does not sacrifice assurance, since it preserves computational divergences somewhere within the checkpoint stream. It merely offers less parallelism opportunities to the verifier by clustering more verification data into fewer checkpoints. A more detailed investigation of the performance trade-offs of this tuning approach is reserved for future work.

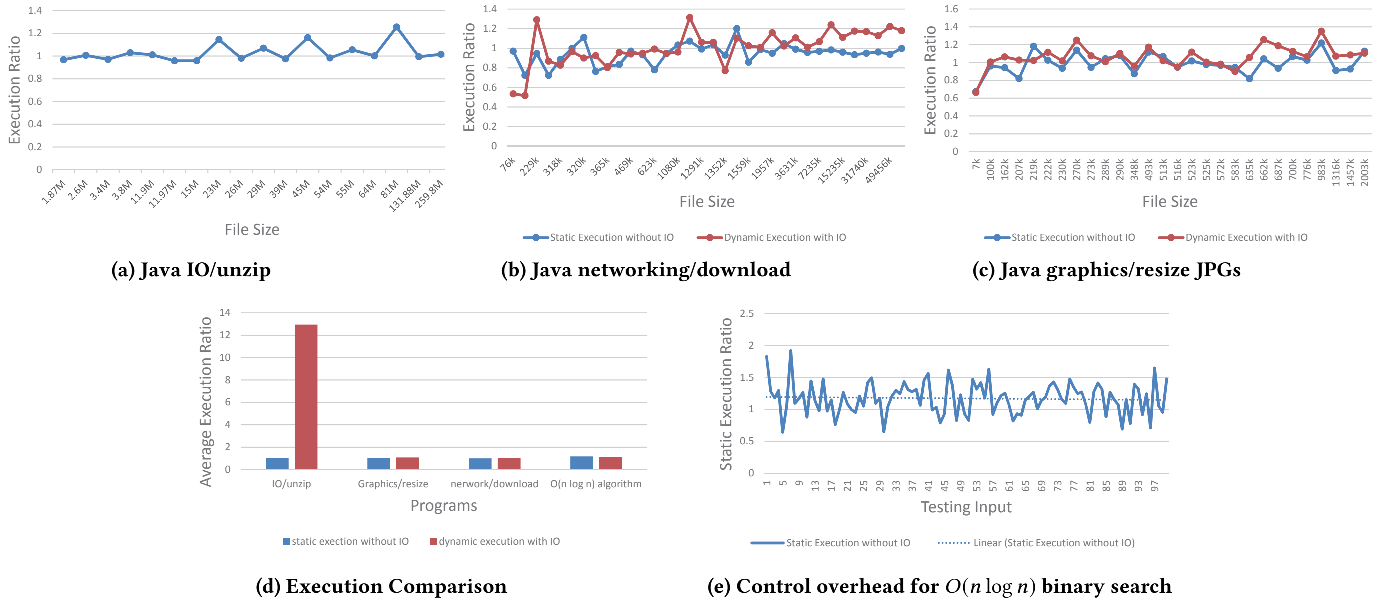
Our experimental data also indicates that input file sizes affect the stability of performance measurements. When file sizes are too big or too small, the performance timers yield unstable outputs. This effect can be seen in Figures 8a and 8b, where the values on the right and left, respectively, fluctuate more widely than in the center. Values at the centers of the plots should therefore be considered more reliable and indicative of real-world observations.

Figure 8d contains an outlier for unzip files, which we investigated in detail to ascertain the cause. It occurs because the test program allocates a large buffer as one of its local variables and access it in an innermost loop, causing J-Gang to include it in many of its checkpoints. The performance could be greatly improved by introducing heuristics whereby J-Gang removes unmodified portions of arrays to its frame state (see §2.5) rather than including them in every checkpoint just becomes some elements were modified. This is another optimization that should be considered by future work.

There is a significant time difference between the dynamic and static modes, especially on experiments with mathematical algorithms. After investigating this, we determined that the higher runtimes reported for dynamic mode are almost entirely due to log access I/O costs that could be significantly improved in a production version of the system. In particular, our prototype stores logs by piping the output of I/O into *System.out* when in-lining the methods in bytecode. All the dynamic test results are therefore

Table 1: Performance evaluation of Java algorithms using J-GANG. *Partial* granularity omits verifying trusted API methods.

Program	Original (ms)	Granu-larity	Static (ms)	Dynamic (ms)	Delay (ns/B)	Log (KB)	Time
binary & sequential search ($n = 100000$)	200.101	partial	211.172	247.248	289.245	163	$O(n \log n)$
matrix multiplication ($n = 100$)	16.741	all	603.304	4098.165	2293.258	17799	$O(n^3)$
2-color algorithm for bipartite ($V = 500$, $E = random(\frac{V^2}{4})$)	2.561	all	68.917	2835.420	1098.829	25709	$O(V + E)$
mode of a set ($n = 1000$)	2.136	all	91.002	1318.665	251.448	52246	$O(n^2)$
all subsets in lexicographic order ($n = 15$)	4.155	all	249.721	4284.198	859.592	49656	$O(2^n \log n)$
nearest neighbor by linear search ($n = 1000$, $D = 2$)	0.078	all	0.463	29.476	1223.429	240	$O(nD)$



(a) Java IO/unzip **(b) Java networking/download** **(c) Java graphics/resize JPGs**

(d) Execution Comparison **(e) Control overhead for $O(n \log n)$ binary search**

Figure 8: Experimental results with utility applications as input

bounded with the delay caused by the console output. In the related testing in the industry, this delay has been shown to be much more expensive than other forms of I/O (e.g., 100× higher than directing I/O to files).[3,4] Although we cannot isolate this portion of the overhead precisely, we can estimate it by artificially inflating the sizes of the log files and observing the relation between runtime overhead and log file size. The results of this analysis indicate that more than 90% of the dynamic mode overhead is due to console I/O. In the independent unit testing on I/O, the average result of delay on I/O for stream to console is around 454ns, which is within the range of the results for the Delay column of Table 1. An obvious next step to improving our prototype implementation is therefore to replace console logging with a high performance filesystem or other storage medium.

In addition, I/O overheads can be further minimized by performing I/O more asynchronously. Doing so avoids delaying the main computation at the cost of slightly increasing the delay between the full-speed CPU computation and the GPU verifier's detection of faults and intrusions. The delay due to I/O latency is only around 0.001 second/KB on average for our prototype.

[3] https://stackoverflow.com/questions/4437715
[4] https://stackoverflow.com/questions/18584809

The influence of memory overhead is minor relative to the I/O overhead, and its scale is determined by the size of input of a program. To avoid a predictable overhead to an uncertain input, it is best to adjust the logging granularity or optimize the tracking by in-lining some trusted methods.

4.2 Verification and Correctness

To verify the correctness, we tested whether our system can detect vulnerabilities of the JVM exploited by flawed or malicious input programs. For accuracy, we only chose the bugs verified by Oracle Java Bug Database. Reproducing the vulnerabilities in Table 2 requires different versions of Jave SE. No simulated program is used in the testing for correctness. The 8 selected bugs are non-duplicated and 7 of them are not related except the second and third.

The vulnerabilities we tested span all officially released subversion of Java SE 6–8. Java SE 9 and 10 are not included because Java 9 non-critical bugs will not be fixed and added in the subversions, and Java 10 was released concurrently with our research. All JVM bugs and test code for them were drawn from Oracle's official bug database. There are usually several bugs (sometimes none) in each subversion that are related to Java official compiler Hotspot based on Windows x86/x64. Among them, we selected bugs that

Table 2: Tested bugs

No.	Bug ID	Description
1	JDK-5091921	Sign flip issues in loop optimizer
2	JDK-8029302	Performance regression in Math.pow intrinsic
3	JDK-8063086	Math.pow yields different results upon repeated calls
4	JDK-8166742	SIGFPE in C2 Loop IV elimination
5	JDK-8184271	Time related C1 intrinsics produce inconsistent results when floating around
6	JDK-7063674	Wrong results from basic comparisons after calls to Long.bitCount(long)
7	JDK-8046516	Segmentation fault in JVM
8	JDK-8066103	Compiler C2's range check smearing allows out of bound array accesses

are testable and offer related source code. While our approach is applicable to vulnerabilities reported elsewhere, such as in malware threat reports, JVM bugs that have not yet been documented in Oracle's official database are extremely difficult to reproduce reliably, and are therefore not tested in this work.

Generally, the eight vulnerabilities listed in Table 2 arise from inaccurate calculations of CPUs in comparison with GPUs. Half of them cause CPUs to perform incorrect floating point computations. Inaccuracies of this form undermine numerous secure computations, such as encryption, related to floating point. Other bugs in the list invite software compromises. For example, testers reported that the false access to arrays caused by JDK-8066103 can be abused to corrupt the heap in ways that victims are unlikely to notice for significant lengths of time. The sign flip problem JDK-5091921 is related to about 30 bug reports and is major facilitator of denial-of-service attacks against Java-based servers.

J-Gang detects all the exploits in Table 2 as a divergence of the CPU and GPU computations. Our testing methodology for confirming this is detailed below.

In each exploit of the 8 vulnerabilities, we first reproduce the exploit to confirm that we have vulnerable execution environment with a proper version of the Java SE and running flags. We then run the code on J-Gang and perform GPU-based validation of the computation. In some cases, we needed to make minor manual adjustments to the proof-of-exploit code to get it to execute, or to keep it compatible with our evaluation infrastructure. None of these manual adjustments affect the exploit itself, or introduce any vulnerability- or exploit-specific mitigations. Manual adjustments needed include the following:

- Some code with new or lesser used Java language features cannot be processed by some of the tool packages underlying our prototype implementation. Such code was adjusted to exclude the unsupported features when the features are not part of the exploit.
- Some exploits become inadvertently corrected merely by the introduction of J-Gang's logging code. For example, the logging code may deactivate a buggy JVM loop optimization. In a real deployment, this is an advantage to defenders since the instrumented code is no longer exploitable. However, to force the exploit to work and test its effect, we manually omitted or moved any checkpoint sites that had the side-effect of fixing the exploit being tested.
- Certain atypical forms of variable assignment, such as reflective updates, are not yet supported by our prototype. We converted such operations to supported equivalents when doing so did not affect the exploit being tested.
- Some proof-of-concept exploit code causes the JVM to freeze instead of hijacking or crashing the victim application. This is typically an artifact of the proof-of-exploit implementation (since real attacks tend to abuse the vulnerability to greater effect). Freezes yield no more checkpoints, so are detectable by timeout rather than by computation divergence. To change freezes into divergences, we artificially force a final checkpoint for such computations.

5 RELATED WORK

5.1 Execution Variance

When *n*-version programming was first introduced in 1978, it opened the field to further improvements in the reliability of software execution, including advantages for fault-avoidance and fault-tolerance [5, 9]. Subsequent experiments identified independence and diversity of software variants as a critical challenge for the approach [30]. In particular, software ecosystems created by independent humans from a common specification exhibit surprisingly low diversity, since humans are prone to making similar mistakes.

In addition, progress in *n*-version programming was severely hindered by the cost of its implementation. Multiple teams of developers were required to build their own version of each piece of software, which was then collected into a single system moderated via a voting strategy to produce results and maintain consistency. This highly manual approach potentially multiplied software development and maintenance costs by a factor of *n*, deterring many practical deployments.

These obstacles motivated automated diversity as a potential amelioration of these dilemmas [12]. Instead of requiring multiple teams, execution diversity can be created by automatically generating variants. Proposed sources of diversity include transformation of nonfunctional code, changing memory layout, and code reordering [17]. For example, prior efforts have maintained and monitored software properties throughout its maintenance lifecycle to help detect when core behaviors could potentially change [55], or have leveraged address space randomization [47] to probabilistically defend against memory errors [8].

The introduction of automation also raised the opportunity to apply *n*-variant programming to address another major rising software problem: cybersecurity. For example, diverse replication was applied to frustrate attempts to hijack operating systems [13]. Within the past decade, this strategy has seen significant progress as software-producing tools, such as compilers, have reached a level of maturity suitable for large-scale, automated *n*-variant deployment (cf., [31]). Recent works have inferred semantics from source code to locate semantic bugs based on multiple different implementations [35], and to build multi-variant execution environments with multi-threading to detect memory corruption vulnerabilities in C/C++ programs [54].

5.2 Heterogeneous Computing

Modern computer programs take advantage of both CPU and GPU components when needed. A survey [36] published in 2015 gives a thorough introduction on this topic. It mentions that one of the motivations for heterogeneous computing is leveraging the unique architectural strength of each processing unit, which corresponds to our idea of utilizing the ability of a GPU to process loops in the program flow. It also introduces hybrid applications and programming languages that span CPUs and GPUs, such as Map-Reduce framework [10, 14, 48, 51] and programming frameworks that eliminate the boundary between CPU and GPU [24, 28, 40, 52]. Map-reduce-like frameworks [24, 28] describe methods for executing source code on both CPUs and GPUs without any modification.

Another way to bridge the differences among hardware is to adopt an intermediate representation. Through this, a program can be automatically dispatched into suitable processing units [40]. For-loop optimizations partition loop iterations across multiple concurrent workers to form a *parallel-for loop* in Java [52]. In our work, the purpose of optimization on the loop is only for verification; so we can evaluate iterations of for-loops in parallel regardless of whether they are computationally parallelizable. This is due to the fact that the CPU replica reveals the (untrusted) input and output states of each iteration in advance.

There are ways to seamlessly develop on GPUs with Java [42]. For example, prior work has applied this technique to translate Java bytecode to OpenCL and implement efficient sample pixel rendering [1]. There are also ways to compile languages into a hybrid environment [18]. Our work does not utilize these approaches since many modifications, including simplification for GPU and optimization, would be required to realize them for the general-purpose computations that we envision as potential subjects of validation.

5.3 Verification

In our work, we consider shrinking the possible state space in the redundant execution since the processing ability of a single processing unit in GPUs is a subset of the CPU computation. Some states must be simplified or canceled, and the verification in our paper is used to describe the assurance of execution results. Through the implementation, we still found some methods to guarantee the quality and scalability for formal verification [15].

To avoid the problem of state space explosion in the procedure of precise verification, multiple strategies can be adopted. One approach is to compress the information of states and still offer explicit checking [23]. Partial order reduction can be used to prune the possible increased space of states [19].

Verification of Java computations is a subject of many prior works (cf., [49]). Java Pathfinder [53] implements model-checking based on an intermediate language [22] to analyze Java bytecode. Primitive types and references are bound to the JVM instructions and incorporated into searches. Type-based abstract interpretation can validate JVM executions [32]. Horn solvers have also been developed for Java verification based on logic programming [29]. Ahead-of-time compilation is another proposed approach [7]. Machine-checked proofs have been constructed to obtain highest possible assurance for Java computations [25], although these approaches currently require significant manual effort.

5.4 Dataflow Analysis

To track inner local variables of methods, our work leverages static dataflow analysis. Such analysis is a staple of program analysis surveyed by numerous prior studies (e.g., [50]). Related works have studied the collection of profiling information in statements via dataflow tracking [2, 6, 43], detection of confidentiality leaks in Java [37], and troubleshooting software errors caused by misconfiguration by tracking dataflows embodying interprocess communications [4].

6 CONCLUSION

This paper proposed and implemented J-Gang, an *n*-variant system framework for verification of Java code by which vulnerabilities can be detected and exposed as the divergence of the execution between CPU and GPU computations. Our solution translates general source code and introduces it into kernels, which yields a solution for executing general Java code in GPUs. To overcome performance disadvantages related to executing mostly serial computations on GPUs, J-Gang leverages GPU parallelism to validate many CPU loop iterations concurrently, affording the GPU variant a means to keep pace with the CPU variant even on computations that are not automatically parallelizable outside of an *n*-variant setting.

We evaluate our system based on the source code of utility applications, known public vulnerabilities, and classic algorithms in Java. A clear security benefit of our work is to detect possible unknown vulnerabilities, including zero-day attacks, while vulnerable programs are executing. Intrusions are detected by the GPU variant on a small delay, whereupon the defense can potentially intervene by raising an alert, aborting the computation, and/or rolling the system back to a safe state.

Prototype implementation of the approach demonstrates significant promise, but exhibits some high overheads for certain operations, such as intensive mathematical computations and high-volume I/O. These observations motivate future work on optimizations that better parallelize nested loops and replace synchronous I/O with asynchronous I/O to improve runtimes.

ACKNOWLEDGMENTS

This work is supported in part by NSF award #1513704, ONR award N00014-17-1-2995, an NSF I/UCRC award from Lockheed Martin, and an endowment from the Eugene McDermott Foundation. Any opinions, recommendations, or conclusions expressed herein are those of the authors and not necessarily of the above supporters.

We thank Dr. Michela Becchi for her valuable suggestions in the revision procedure.

REFERENCES

[1] Razvan-Mihai Aciu and Horia Ciocarlie. 2016. Runtime Translation of the Java Bytecode to OpenCL and GPU Execution of the Resulted Code. *Acta Polytechnica Hungarica* 13, 3 (2016), 25–44.

[2] Hira Agrawal. 1999. Efficient Coverage Testing Using Global Dominator Graphs. In *Proc. ACM SIGPLAN-SIGSOFT Work. Program Analysis for Software Tools and Engineering (PASTE).* 11–20.

[3] Yousra Alkabani and Farinaz Koushanfar. 2008. N-variant IC Design: Methodology and Applications. In *Proc. 45th ACM/IEEE Design Automation Conf. (DAC).* 546–551.

[4] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proc. USENIX Sym. Operating Systems Design and Implementation (OSDI).* 237–250.

[5] Algirdas Avižienis. 1985. The N-version Approach to Fault-tolerant Software. *IEEE Trans. Software Engineering (TSE)* 11, 12 (1985), 1491–1501.

[6] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *Proc. 29th Annual ACM/IEEE Int. Sym. Microarchitecture (MICRO)*. 46–57.

[7] James Baxter. 2017. *An Approach to Verification of Safety-critical Java Virtual Machines with Ahead-of-time Compilation.* Technical Report. University of York.

[8] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. 2005. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proc. 14th USENIX Security Sym.*

[9] Liming Chen and Algirdas Avižienis. 1978. N-version Programming: A Fault-tolerance Approach to Reliability of Software Operation. In *Proc. IEEE Int. Conf. Fault-tolerant Computing (FTCS)*. 3–9.

[10] Linchuan Chen, Xin Huo, and Gagan Agrawal. 2012. Accelerating MapReduce on a Coupled CPU-GPU Architecture. In *Proc. 24th Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC).*

[11] Shigeru Chiba. 2000. Load-time Structural Reflection in Java. In *Proc. 14th European Conf. Object-oriented Programming (ECOOP)*. 313–336.

[12] Fred Cohen. 1993. Operating System Protection Through Program Evolution. *Computers and Security* 12, 6 (1993), 565–584.

[13] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. 2006. N-variant Systems: A Secretless Framework for Security Through Diversity. In *Proc. 15th USENIX Security Sym.*

[14] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Communications ACM (CACM)* 51, 1 (2008), 107–113.

[15] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. 2008. A Survey of Automated Techniques for Formal Software Verification. *IEEE Trans. Computer-aided Design Integrated Circuits and Systems (TCAD)* 27, 7 (2008), 1165–1178.

[16] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 2002. A Programmer's Reduction Semantics for Classes and Mixins. In *Formal Syntax and Semantics of Java*. Springer, 241–269.

[17] Stephanie Forrest, Anil Somayaji, and David H. Ackley. 1997. Building Diverse Computer Systems. In *Proc. 6th Work. Hot Topics in Operating Systems (HotOS)*. 67–72.

[18] Rahul Garg and José Nelson Amaral. 2010. Compiling Python to a Hybrid Execution Environment. In *Proc. 3rd Work. General-purpose Computation Graphics Processing Units (GPGPU)*. 19–30.

[19] Patrice Godefroid. 1996. *Partial-order Methods for the Verification of Concurrent Systems: An Approach to the State-explosion Problem.* Springer-Verlag, Berlin, Heidelberg.

[20] Brian Goetz. 2014. State of the Specialization. http://cr.openjdk.java.net/~briangoetz/valhalla/specialization.html. (December 2014).

[21] Axel Habermaier. 2011. *The Model of Computation of CUDA and its Formal Semantics.* Technical Report 2011-14. Institut für Informatik, U. Augsburg.

[22] Klaus Havelund and Thomas Pressburger. 2000. Model Checking JAVA Programs Using JAVA PathFinder. *Int. J. Software Tools for Technology Transfer (STTT)* 2, 4 (2000), 366–381.

[23] Gerard J. Holzmann. 1997. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In *Proc. 3rd Int. SPIN Work.*

[24] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. 2010. MapCG: Writing Parallel Program Portable Between CPU and GPU. In *Proc. 19th Int. Conf. Parallel Architectures and Compilation Techniques (PACT)*. 217–226.

[25] Thierry Hubert and Claude Marché. 2005. A Case Study of C Source Code Verification: The Schorr-Waite Algorithm. In *Proc. 3rd IEEE Int. Conf. Software Engineering and Formal Methods (SEFM)*. 190–199.

[26] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Programming Languages And Systems (TOPLAS)* 23, 3 (2001), 396–450.

[27] Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz. 2011. Compiler-generated Software Diversity. In *Moving Target Defense.* Springer, 77–98.

[28] Wei Jiang and Gagan Agrawal. 2012. MATE-CG: A Map Reduce-like Framework for Accelerating Data-intensive Computations on Heterogeneous Clusters. In *Proc. IEEE 26th Int. Parallel and Distributed Processing Sym. (IPDPS)*. 644–655.

[29] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. 2016. JayHorn: A Framework for Verifying Java Programs. In *Proc. 28th Int. Conf. Computer Aided Verification (CAV)*. 352–358.

[30] John C. Knight and Nancy G. Leveson. 1986. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Trans. Software Engineering (TSE)* 12, 1 (1986), 96–109.

[31] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *Proc. 35th IEEE Sym. Security & Privacy (S&P)*. 276–291.

[32] Xavier Leroy. 2003. Java Bytecode Verification: Algorithms and Formalizations. *J. Automated Reasoning* 30, 3 (2003), 235–269.

[33] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Communications ACM (CACM)* 52, 7 (2009), 107–115.

[34] Ogier Maitre. 2013. Understanding NVIDIA GPGPU Hardware. In *Massively Parallel Evolutionary Computation on GPGPUs*, Shigeyoshi Tsutsui and Pierre Collet (Eds.). Springer, 15–34.

[35] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *Proc. 25th Sym. Operating Systems Principles (SOSP)*. 361–377.

[36] Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys (CSUR)* 47, 4 (2015).

[37] M. Mongiovì, G. Giannone, A. Fornaia, G. Pappalardo, and E. Tramontana. 2015. Combining Static and Dynamic Data Flow Analysis: A Hybrid Approach for Detecting Data Leaks in Java Applications. In *Proc. 30th Annual ACM Sym. Applied Computing (SAC)*. 1573–1579.

[38] Anh Nguyen-Tuong, David Evans, John C. Knight, Benjamin Cox, and Jack W. Davidson. 2008. Security Through Redundant Data Diversity. In *Proc. IEEE Int. Conf. Dependable Systems and Networks (DSN)*. 187–196.

[39] Emma Nilsson-Nyman, Görel Hedin, Eva Magnusson, and Torbjörn Ekman. 2008. Declarative Intraprocedural Flow Analysis of Java Source Code. In *Proc. 8th Work. Language Descriptions, Tools and Applications (LDTA)*. 155–171.

[40] Sreepathi Pai, Ramaswamy Govindarajan, and Matthew Jacob Thazhuthaveetil. 2010. PLASMA: Portable Programming for SIMD Heterogeneous Accelerators. In *Proc. Work. Language, Compiler, and Architecture Support for GPGPU.*

[41] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. SPOON: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46, 9 (2015), 1155–1179.

[42] Philip C. Pratt-Szeliga, James W. Fawcett, and Roy D. Welch. 2012. Rootbeer: Seamlessly Using GPUs From Java. In *Proc. IEEE 14th Int. Conf. High Performance Computing and Communication (HPCC) & IEEE 9th Int. Conf. Embedded Software and Systems (ICESS)*. 375–380.

[43] Sandra Rapps and Elaine J. Weyuker. 1985. Selecting Software Test Data Using Data Flow Information. *IEEE Trans. Software Engineering (TSE)* 11, 4 (1985), 367–375.

[44] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. 2009. Orchestra: Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-space. In *Proc. 4th ACM European Conf. Computer Systems (EuroSys).*

[45] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proc. 36th IEEE Sym. Security & Privacy (S&P)*. 745–762.

[46] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proc. 14th ACM Conf. Computer and Communications Security (CCS)*. 552–561.

[47] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-space Randomization. In *Proc. 11th ACM Conf. Computer and Communications Security (CCS)*. 298–307.

[48] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. 2010. Hybrid Map Task Scheduling for GPU-based Heterogeneous Clusters. In *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science (CloudCom)*. 733–740.

[49] Robert F Stärk, Joachim Schmid, and Egon Börger. 2012. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer Science & Business Media.

[50] Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. 2017. A Survey on Data-flow Testing. *ACM Computing Surveys (CSUR)* 50, 1 (2017).

[51] Kuen Hung Tsoi and Wayne Luk. 2010. Axel: A Heterogeneous Cluster with FPGAs and GPUs. In *Proc. 18th Annual ACM/SIGDA Int. Sym. Field Programmable Gate Arrays (FPGA)*. 115–124.

[52] Ronald Veldema, Thorsten Blass, and Michael Philippsen. 2011. Enabling Multiple Accelerator Acceleration for Java/OpenMP. In *Proc. 3rd USENIX Conf. Hot Topic in Parallelism (HotPar).*

[53] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. 2004. Test Input Generation with Java PathFinder. In *Proc. ACM SIGSOFT Int. Sym. Software Testing and Analysis (ISSTA)*. 97–107.

[54] Stijn Volckaert, Bart Coppens, Bjorn De Sutter, Koen De Bosschere, Per Larsen, and Michael Franz. 2017. Taming Parallelism in a Multi-variant Execution Environment. In *Proc. 12th European Conf. Computer Systems (EuroSys)*. 270–285.

[55] Jinlin Yang and David Evans. 2004. Automatically Inferring Temporal Properties for Program Evolution. In *Proc. 15th Int. Sym. Software Reliability Engineering (ISSRE)*. 340–351.