Average Case vs. Worst Case—Margins of Safety in System Design

Christian W. Probst
Informatics and Mathematical Modelling
Technical University of Denmark
2800 Kongens Lyngby, Denmark
probst@imm.dtu.dk

Andreas Gal Michael Franz
Donald Bren School of Information and
Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
{gal,franz}@uci.edu

ABSTRACT

We predict that we will soon witness attacks on all kinds of systems that will be based on the attacked systems' worst-case behavior. For example, the worst-case performance of Java Bytecode Verification rises quadratically with program length. By sending a legal, but difficult-to-verify program to a server virtual machine, we can keep that server occupied for an inordinate amount of time, effectively making it unavailable for useful work. The problem, however, is not restricted to mobile-code verification: for example, an attacker could exploit knowledge about a just-in-time compiler's register allocator by sending it a particularly difficult to solve graph-coloring puzzle. The same vulnerability can be exploited if the attacker has intimate knowledge of the data structures used in the attacked system. Similar problems occur in hardware, e.g. with respect to power variability or the heat dissipation of processors. Malicious programs can exploit which parts of computer chips dissipate power, thereby overheating regions of the chip that are known to contain no temperature sensors. This attack could be used to affect battery life or cause early chip aging. Unfortunately, worst case-based attacks are hard to counter without also limiting the system's behavior in the average case.

1. INTRODUCTION

Recently, first attacks based on the worst-case behavior of systems have been reported—in areas as different as mobile code [12], general applications [8], and heat dissipation of processors [9]. The main characteristic shared by these attacks is that they target systems that have been optimized for the *average* case and do not include sufficient margin of safety for the *worst* case.

In this paper, we contend that correct behavior in the average case is not sufficient to defend systems against worst-case behavior based attacks. Instead, there needs to be a dual focus on the (worst-case-)*performance*. Otherwise, as we will show, systems become vulnerable. Often this means that systems need to be re-designed to prevent such attacks in the first place.

For example, the JVM bytecode verification algorithm exhibits quadratic worst-case execution complexity. We have been able to construct relatively small mobile programs that require hours of verification on high-end workstations. The programs in question are perfectly legal JVM code, perform no malicious action on the host, and will eventually be verified as being safe. However, the process of verification itself is so costly as to effectively constitute a denial-of-service attack. Current mobile code systems treat verification and just-in-time code generation as atomic operations. As far as we know, there is not a single existing Java Virtual Machine in which verification or just-in-time code generation can be interrupted by the user—other than by killing the whole Virtual Machine. And for the increasingly important *server-side virtual machines*, human intervention is not even an option.

The problem with these kind of attacks is that the scenarios where the attack occurs do not constitute "illegal" uses of the system being attacked. In fact, there might be completely reasonable *valid and useful* programs or inputs, that just by incident—or by malicious intent—cause the system to exhibit some kind of worst-case behavior. There often exist simple precautions against these attacks. In the case of bytecode verification, e.g., one could deploy a traditional monitor that would abort verification when a certain time limit is exceeded. It is noteworthy, however, that these countermeasures always come with the risk of rejecting certain useful programs.

Hence, the problem is similar to that of defending against low-intensity viruses or worms that cause damage while "flying under the radar" without ever tipping off an intrusion-detection system. Unlike such a low-intensity virus or worm, however, the attack in our case consumes all of the cycles of the host and prevents useful calculations.

Unfortunately, attacks based on worst-case behavior can affect *all* parts of systems. In the compilation pipeline of Mobile Code Execution Frameworks, e.g., an adversary that knows the target virtual machine's register allocation algorithm might be able to maliciously craft a valid mobile-code program containing a particularly difficult to solve graph-

NSPW 2005 Lake Arrowhead CA USA

coloring puzzle. Or, to attack the heat sensors present on modern processors, an adversary could send a program that specifically over-uses parts of the chip that are not close to sensors. This kind of attack results in an overheating of parts of the chip, thereby causing, e.g., increased power consumption or accelerated chip aging [9].

An interesting point to note is that "security by obscurity" seems to be a perfectly valid approach to secure systems against this kind of attacks. However, in the end it only increases the effort needed to obtain the necessary information to run an attack. In the case of systems whose source code is available this effort is basically not existent, for binaries and hardware it consists of re-engineering the system behavior and exploring its construction, and for systems for which only the input/output behavior can be observed the effort consists in analyzing exactly behavior.

In the realm of hardware, "security by obscurity" is applied by default. Since, e.g, the on-chip temperature sensors are hidden in the chip packaging, there is no easy way to find out whether the chip only has sensors as described in the specification. As we will discuss briefly in Section 2.2.2, chips might very well contain more heat sensors than documented, to limit the chip's vulnerability.

The remainder of this paper is structured as follows: The next section gives several examples for systems that have *not* been designed with the worst case in mind. The examples chosen come from two areas, namely the vulnerability and attackability of data operations and hardware systems. These examples support our contention that a paradigm shift is necessary towards making systems aware of worst-case behavior based attacks and hardening them against those attacks. In Section 3, we argue that the solution may very well lie in designing systems based on their *worst-case*, rather than *average-case* behavior. Currently, systems like virtual machines, JIT compilers, and many processors are fine tuned for either high performance or low cost in the average case, with the hope that the worst case will rarely occur—we assert that this approach is simply too dangerous. A summary concludes our paper (Section 5).

2. WORST CASE BEHAVIOR-BASED ATTACKS

Systems that accept user programs or inputs must use special care to avoid that the input causes one or several system components to exhibit worst-case behavior. This section presents some examples for such systems that must be optimized for the *worst* and not for the *average* case. We start by looking at attacks based on some form of algorithmic complexity (Section 2.1), and later on present an example from the realm of hardware (Section 2.2). A commonality of the attacks described is that in order to succeed, the attacker needs intimate knowledge about the implementation or the system layout.

Beside the examples presented, there are many more documented in literature. Garfinkel [15] describes nested HTML tables as an attack on some browser. Due to bad implementation of the layout algorithms in some browsers, they would perform super-linear work to determine the on-screen layout of the table. Stubblefield and Dean [10] describe an attack against SSL servers. In this attack a malicious client coerces a web server into performing expensive RSA decryption. Again this attack is based on the intimate knowledge of the implementation of the web server.

```
$todo \leftarrow true$
while $todo = true$ do
  $todo \leftarrow false$
  for all $i$ in all instructions of a method do
    if $i$ was changed then
      $todo \leftarrow true$
      check whether stack and local variable types
      match definition of $i$
      calculate new state after $i$
      for all $s$ in all successor instructions of $i$ do
        if current state for $s \neq$ new state derived from
        $i$ then
          assume state after $i$ as new entry state for $s$
          mark $s$ as changed
        end if
      end for
    end if
  end for
end while
```

Figure 1: The standard verification algorithm found in Sun Microsystem's JVM implementations.

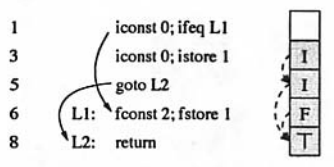
2.1 Algorithmic Complexity Attacks

In this section we look at three examples from different areas, that are closely related. We start with a denial-of-service attack on the Java Bytecode Verifier, based on the knowledge how a large class of Virtual Machines perform verification. The next attack applies the same ideas to the compilation pipeline in mobile-code execution frameworks. Finally, we describe the general case that the first two examples are specializations of, namely the attackability based on the worst-case behavior of operations on data structures.

2.1.1 Java Bytecode Verification

Static mobile code verification was introduced as an alternative to the *dynamic* checking of type safety properties at runtime through dynamic execution monitors [7] by Gosling and Yellin [28, 21]. Using dynamic runtime checks can cause a significant runtime overhead at execution time. The basic ingredient of every JVM bytecode verifier is an abstract interpreter for Java Virtual Machine Language instructions. The stacks and virtual registers of this abstract interpreter store *types*, rather than *values*. Similarly, the instructions of the abstract interpreter operate at the type-level only and do not perform any actual calculations. For an extensive discussion of conditions for bytecode to be accepted by the verifier see [20]. Figure 1 shows a simplified version of the algorithm that is used, with slight modifications, in all Java Virtual Machine implementations that we are aware of, including Sun's own CVM [27] and HotSpot virtual machines [26].

Regarding the complexity of verification, the analysis of straight-line code is inexpensive, since the abstract interpreter only propagates type information through the instructions and computes the abstract stack state after each in-



```
1           iconst 0; ifeq L1
3           iconst 0; istore 1
5           goto L2
6     L1:   fconst 2; fstore 1
8     L2:   return
```

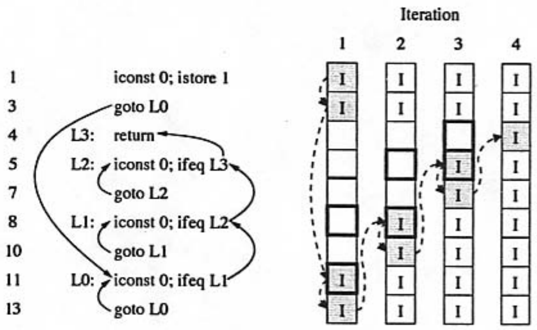
Figure 2: Verification of Java byte-code through iterative data-flow analysis. The verifier traverses the method from the first instruction to the last. While conditional branch instructions such as *ifeq* are either taken or not-taken by the virtual machine, the abstract interpreter considers both cases at the same time. In this example, the local variable is set to a float in one of the branches, and to an integer in the other. At the merge point (instruction 8), the type of the variable becomes ⊤, because the type of the local variable depends on whether the branch was taken or not. Any attempts by the program to read local variables of type ⊤ would be rejected by the verifier. The example code shown here contains no backward branches, and hence the analysis can be completed in a single iteration. If the taken and not-taken code blocks had been located *before* the *ifeq* instruction (backward branch), the abstract interpreter would have had to iterate over the code a second time to determine the type of the local variable in the merge point.

struction. The runtime of such a data-flow analysis is significantly increased if the code contains jumps, exception handlers, or subroutines, which introduce forks and joins in the control-flow graph. When separate control flows are merged together, an instruction's predecessors may have different abstract stack or variable types. After merging the state information of the two incoming control flows, the data-flow analysis has to be repeated for all instructions which are reachable from this point in the control flow of the method. For simplicity, the existing Java verifier repeats the entire data-flow analysis for every instruction of a method until there are no more changes.

For average Java programs, the verifier algorithm quickly reaches a fixed point after only a few iterations. For straight-line code or code that contains only forward branches, the verification algorithm terminates already after a single iteration (Figure 2). It is obvious that—*in theory*—the Java verifier could need up to n iterations over the method, with n being the number of instructions in the method. Since for each iteration the verifier might have to visit all instructions, the overall complexity is at least $O(n^2)$.

Such quadratic runtime behavior does not only exist in theory. In fact, simple Java programs can be constructed that expose the worst-case scenario in practice. Figure 3 shows a very simple Java program that does nothing but store an integer into a local variable and jump backwards through the code until it finally returns.

Studying the verifier algorithm reveals that newly computed type information is forwarded immediately to instructions that come syntactically *after* the current instruction. To



```
1           iconst 0; istore 1
3           goto L0
4     L3:   return
5     L2:   iconst 0; ifeq L3
7           goto L2
8     L1:   iconst 0; ifeq L2
10          goto L1
11    L0:   iconst 0; ifeq L1
13          goto L0
```

Figure 3: Java bytecode program that takes n iterations to be verified using Sun's standard DFA verifier approach. The entry state for each basic block depends on the successor basic block. The type of the first local variable is displayed for each iteration of the DFA. It is initially assumed to be of unknown type and is discovered to be an integer (I) during successive iterations. Shaded boxes indicate a change in the current iteration, framed boxes will be visited in the next iteration.

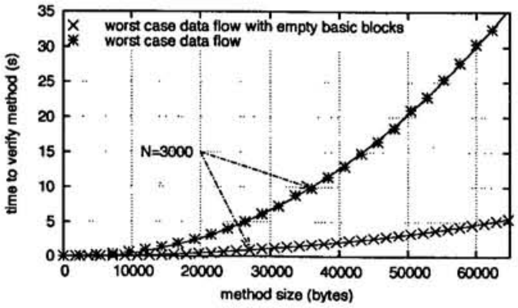


Figure 4: Verification time for verifying a single method containing a worst-case data-flow scenario. The x-axis indicates the length of the method bytecode in bytes, which is proportional to the number of basic blocks N used to construct the code. The arrows indicate for comparison purposes the code size for path length $N = 3000$.

instructions that come syntactically *before* the current instruction, the new abstractions will only be forwarded in the next iteration of the DFA. The simplistic approach of the traditional Java bytecode algorithm to iterate over the bytecode until a fixed point is reached simplifies the generation of attacks like the one shown in Figure 3, but any other iteration order would also exhibit a particular (possibly different) worst-case behavior for which a malicious program could be constructed.

We have measured the verification time for two malicious programs designed to exhibit the worst-case performance of the Java verifier using the Sun Microsystems Java 2 HotSpot Client VM [12]. Figure 4 shows the verification time for a single method containing bytecode with an increasing maximum data-flow path of length N. This time includes only the time it takes the verifier to prove safety. The code

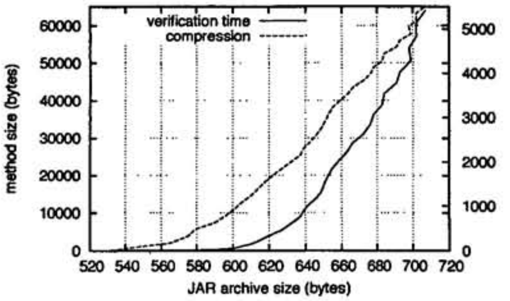


Figure 5: Compression of constructed code examples using the standard JAR archive format. The code is extremely well compressible as it repeats identical code patterns. While the verification times increases by over factor 5000, the JAR file merely grows by less than 200 bytes.

is never actually executed or compiled to executable code. The first curve shows the verification time for a worst-case path length problem with empty basic blocks. The second curve in the graph shows the maximum flow path problem with some additional code added to each basic block, which further slows down the verifier. Both curves clearly show quadratic growth.

All measurements were taken on a 2.53 GHz Pentium 4 and the Sun HotSpot VM 1.41. The maximum verification time we observed on this machine *for a single method* was approximately 40 seconds. Since the size of method code in Java is limited, this time can not be increased. However, to achieve even longer verification times, an attacker could hide more than just one of these methods in the code. Just including 20 methods instead of one already increases the verification time to approximately 15 minutes on the test machine we used.

The standard JAR archive format used by Java can be used to drastically reduce the apparent size of the malicious code. The code patterns used in the presented scenarios lend themselves for compression due to their very regular structure. Figure 5 indicates the compressed size for different problem lengths N. While the verification times increases by over factor 5000, the JAR file merely grows by less than 200 bytes. The JAR archive format thus represents another example of a well-meant algorithm with appropriate average-case performance, which however exhibits very unexpected worst-case behavior.

We have used the two algorithmic shortcomings described here to construct a malicious applet [11] that disables the Java VM of web browsers for some time. The applet is 10kb in size and indistinguishable from regular applet code, because it is a legal and correct Java program. Short of disabling Java applets, the user cannot prevent or interrupt the loading of this applet. In fact, existing browsers do not even allow the user to interrupt the verification because the browser implementor never considered the verification time to be costly enough. Some browsers, including some versions of the Microsoft Internet Explorer, allow the verifier to continue the verification silently and continue to hog the CPU in the background even if the user leaves a website containing an applet that takes an excessive amount of time to verify.

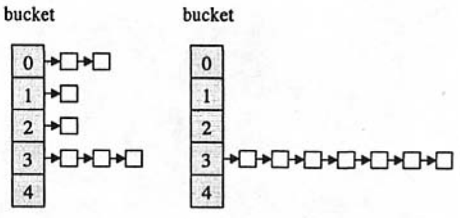


Figure 6: Normal operation of a hash table implementation (right hand side) versus collisions caused by, e.g., malicious input.

2.1.2 *Attacking the Mobile-Code Compilation Pipeline*

Denial-of-service attacks are not limited to the bytecode verification phase, which is executed early in a bytecode-execution framework. Any code transformation algorithm applied to mobile code during its path from a portable bytecode format to natively executable machine code is vulnerable at its point of worst-case complexity. This applies in particular to compiler optimization algorithms, which are traditionally chosen for speed in the average case but not for worst-case performance, and some of which use heuristics to solve problems like graph coloring and instruction scheduling that are known to be NP-complete [6, 17].

An example for such an attackable optimization algorithm is register allocation. Register allocation is an important component of any JIT compiler that strives to achieve good code quality. The classic register-allocating algorithm is structured after Chaitin's graph coloring allocator [6, 5]. Many improvements and variants have been proposed [2, 3, 16, 19], but most of this research was focused on improving the average-case performance. Poletto et al. showed that register allocation using graph-coloring has a quadratic worst-case complexity for certain pathological cases [23] and proposed a linear-scan algorithm for register allocation. This algorithm is not guaranteed to find the optimal register allocation for any given problem, but has a linear worst-case performance. To truly harden the virtual machine against worst-case behavior based denial-of-service attacks, however, this principle of trading off some code quality in return for linear time complexity has to be extended to the entire code-processing pipeline.

2.1.3 *Attacking Data-Structure Operations*

If we abstract from the just presented examples, the common property is the worst-case behavior of the underlying *algorithms*. Crosby and Wallach [8] show how to attack systems for which the implementation of certain data structures like hash tables is known. Hash tables have an average-case complexity of $O(n)$ for inserting n elements, and a worst-case complexity of $O(n^2)$ if all elements hash to the same bucket in the table. Figure 6 shows the comparison between the average case and the worst case as given by [8].

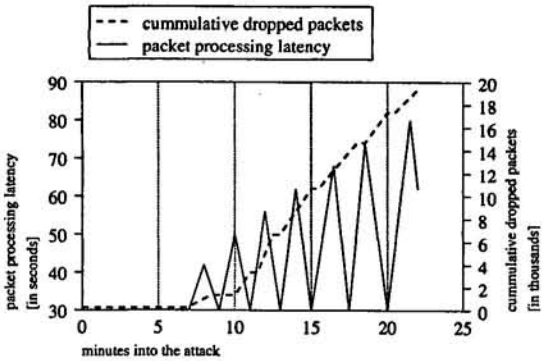


Figure 7: Performance of the network intrusion detector Bro under attack. The solid line marks the latency for processing of received packets, the dashed line the number of dropped packets in thousands. The attack is based on sending SYN packets at a rate of 16kb/second.

Crosby and Wallace describe how to compute such an attack on hash tables [8]. They also describe attacks on two hash table implementations of the Perl interpreter, the Squid web proxy, and the DJB DNS cache. Finally, they present an attack on the Bro intrusion detection system [22], which also is highly vulnerable to the proposed attacks. Figure 7 shows the result of attacking Bro with SYN packets at a rate of 16kb/second. As can be seen, already for this relatively slow attack the number of dropped packages and the latency in processing received packages is considerable. For more details on the attacks c.f. [8].

Obviously this kind of attack is not limited to hash tables, but in principle can be used against any data structure and its operations that fulfill three conditions: there must be at least an order of magnitude difference between the runtime for the average and the worst case, the operations must be deterministic, and the source code must be available.

2.2 Hardware Attacks

This section complements the just described attacks on software with two examples for attacks on hardware. We start by describing an attack based on the behavior of a power supply subsystem when the current drawn by the system changes. The second attack targets chips and their on-chip heat sensors. Compared to many software systems, hardware usually has the advantage that

2.2.1 Power Variability Attacks

Joseph et al. [18] describe a possible attack caused by the increasing focus on power dissipation issues in current microprocessors. These issues have lead to a group of proposals of power-saving techniques, e.g. clock gating, that generally are very effective in reducing *average* power. However, many of these techniques also result in increased variability of both power dissipation and the current drawn by the processor. This increased variabilities can cause supply voltage fluctuations, which is a significant problem since chips may malfunction if the supply voltage rises or drops out of a chip-specific tolerance range. The variability is caused by the power supply's inductance, which together with the current variations produces jitters on the chip's supply lines. This problem is know as the *dI/dt problem*, since the magnitude of voltage ripples caused depends on the change of current over time.

2.2.2 Heat Dissipation Attacks

Another example for a system that is vulnerable to worst-case behavior based attacks is the cooling system for processors (or chips in general). Obviously, a chip's power and heat dissipation depend on the program(s) executed on the chip. Consequently, a malicious program might try to overheat parts of the chip to cause, e.g., increased power usage to drain batteries, heat damage to the chip, or general instability of the system.

Together with the increase in performance of modern chips comes an increase in power density. To allow high performance while keeping cooling cost low, the cooling system is optimized for the average case instead of the worst case. As the average workload does not induce worst-case power dissipation, especially not over longer periods, the cooling system can be kept much smaller. To guard the system against the worst-case, it then needs to be equipped with sensors to throttle the system if the on-chip heat gets to high, or even shut off or reboot in extreme situations.

The vulnerability caused by this has been reported by Dadvar and Skadron [9]. The authors deal with the Pentium 4, a chip that employs two on-chip sensors to measure heat. The chip's thermal control circuit uses an internal thermal diode and compares it to a reference current. This sensor has been placed close to the area of the chip that is *expected* to be the hottest under *normal* operation. This means that under certain circumstances, namely a workload that does not represent the average case, other regions of the chip might actually become hotter than the area monitored by the sensor. The sensor's measurements are not visible from the outside, but are used exclusively by the thermal control system. Whenever this system detects thermal stress, the CPU activity is throttled by interleaving short periods of complete inactivity with normal operation.

The authors report on their early findings with respect to thermal vulnerabilities. Under normal operating conditions they were not able to cause thermal throttling, however with partial blocking of the system's air vents or with disabling the fans altogether, [1] they could slow down a system by 50%. Even more important, the new multi-threaded Prescott core is reported to reach core temperatures that already during normal operation get close to the throttling trigger temperature. As the authors argue in [9] this will pose a serious risk of thermal attacks against such systems.

3. COUNTERMEASURES—AND WHY THEY DO NOT WORK

In general, the best countermeasure against the attacks described in this paper is to design systems for the worst case—

[1] This requires additional software and user-rights to access the corresponding flags in the system, however the authors report on several programs that give "normal" user programs exactly these rights.

if it is known. In contrast to security flaws previously discovered in systems like the JVM [4], the enabling property for worst-case behavior based attacks on systems is an *inherent property* of the system and not merely some *faulty* implementation or mis-design that could easily be exchanged.

In the case of the JVM verifier, rewriting the algorithm to iterate over the code in some other order, or the introduction of a work list algorithm, would not significantly improve the situation. Each of these algorithms would still expose quadratic runtime behavior for some worst case scenarios.

However, a number of mitigating factors exist. First, current JVMs limit the code size per method to 65,536 bytes. On high-end desktop systems this limits the maximum verification time we were able to achieve using a single method to approximately 40s. This (probably accidental) ceiling prevents the construction of worst case scenarios with near-infinite verification time.

Further shortening the maximum method length of Java methods is not an option, since long Java methods are not uncommon. Some compilers emitting Java bytecode generate long methods close to the limit defined in the Java specification. It would be not surprising if Sun decided to remove the current code size limitation in future versions of the Java Virtual Machine.

It seems unlikely that one could establish a clear set of rules to detect classes of malicious input that are responsible for causing a system to exhibit its worst-case behavior. For the attacks on software systems presented in this paper, such rules could be to reject programs because they take more than a certain number of iterations to verify or because more than a certain number of entries map to the same bucket in a hash table. Obviously, any such number would be chosen arbitrarily and would impose a very vague and imprecise restriction of acceptable programs.

On the other hand, trying to detect patterns such as the ones described in this paper would not eliminate the problem—more complex and less obvious examples can be easily constructed. It would also get us back to the *pattern matching* approach used in virus detection tools, something that bytecode verification was supposed to free us from.

The impact of the complexity-based attacked just described can be increased increasing the intensity of attacks, e.g. by shipping a large number of malicious methods to the verifier, performing several voltage/current changes in short time, or starting several threads with stress marks in short time.

4. A NEW SECURITY PARADIGM

As already pointed out, we contend that the only chance to counter attacks that are based on the worst-case behavior of certain parts of systems is a new security paradigm. Instead of targeting only the safety of certain properties of incoming data, the new paradigm must also take into account the complexity and design of the whole system. In the case of the hardware examples presented in this paper this means that the system must be equipped with sensors to identify the effects of executing malicious programs. For the software examples this means to target the whole compilation path from verification up to execution and the careful design and selection of *all* data structures and operations.

We are currently investigating possible approaches to harden systems against worst-case behavior based attacks. In the case of attacks based on algorithmic complexity, the vulnerability demonstrates the need for not only *correct* but also *efficient* algorithms. With software applications, e.g., moving to Grid- and service-based architectures, in which computations are sent to hosts for execution, these efficient algorithms are going to be essential for system reliability in the near future.

As we have shown previously, the code compression format used by Java lends itself to conceal from the user the true size of transported programs. Compression algorithms can also be exploited in many other ways. Clasen used a missing range check in the zlib decompression algorithm to construct PNG images that crash the browser because the decompression algorithm tries to allocate unreasonably large amounts of memory [24]. It is entirely possible that similar vulnerabilities exist in any other compression format, but this has apparently not yet been studied.

Our own main interest is to harden Java Virtual Machines against the kind of attacks described. Therefore, we are currently constructing an "algorithmic testbed" Java Virtual Machine that can be configured with different variants of critical algorithms. We have also developed a tool to automatically generate JVM class files that present particular hard to solve algorithmic puzzles. This tool is currently used in benchmarking existing JVMs, highlighting their potential vulnerabilities, and aiding the removal of such vulnerabilities. Our aim is to harden the existing Java-based information infrastructure already deployed against such worst-case behavior based attacks. Although no such attacks have yet been reported, they could be very costly in scenarios in which computations are sent to remote servers in the form of "agents".

The scope of this process is quite broad by nature: For many code optimizations, well known heuristics exist to speed up their average case performance. However, little to no emphasis has been placed on the worst-case behavior of these algorithms in the context of being a potential security risk. In particular, iterative analyses such as escape analysis, register coalescing, live-range splitting, instruction scheduling, and register allocation through graph coloring can have a very poor worst-case performance. Existing JIT implementations must be analyzed to identify their weaknesses, and also to provide a framework of code-optimization algorithms with well understood worst-case behaviors.

For the verifier, we have developed such a hardened algorithm. After performing an initial type check using a superficial type system, it converts the Java bytecode to Static Single Assignment form (SSA) [25, 1], and only then checks the consistency of type flows using the whole Java type system to verify type safety [13, 14]. While this algorithm has a higher *average-case* cost than the standard Java verification algorithm, it has a much better *worst-case* behavior. Namely, all phases beside the SSA construction can be performed in linear time. Many higher-end JIT compilers for

Java generate SSA anyway at later stages of dynamic code generation. While SSA construction is the main cost in our algorithm, these frameworks can get verification at an *incremental* cost by using our verifier and reusing the constructed SSA. Currently, they perform the standard Java verification *before* starting the actual compilation.

5. CONCLUSION

Future software-application architectures are moving to service-based and Grid architectures, in which computations are sent to hosts for execution. Soon, these service-based execution frameworks will be omni-present, making the actual network-based execution mechanism invisible to the user. In these architectures, efficient algorithms for each step in the chain from *receiving mobile code* to *compiling it to native code* and *executing it* will be needed to protect against complexity-based attacks. The threat of these subtle denial-of-service attacks has been neglected, apparently because it does not occur in daily average-case use of mobile code. In the case of an unsupervised server at the heart of a service-based framework, however, having the framework verifying, analyzing, compiling, and executing several mobile-code programs in parallel will make each and every phase in the framework vulnerable to complexity-based attacks.

At the same time chips are being optimized to use as little power as possible and their cooling systems are minimized to be only as big as necessary. As a result, large groups of systems are vulnerable by thermal attacks based on power variations and too small cooling systems.

We therefore advocate a new security paradigm based on complexity-hardened systems. Given that currently a large amount of vulnerable systems is already in place, there is no quick fix to this problem. Instead, we will need to rethink the architecture of those systems—while current systems have been selected and designed for their average case behavior, we will need to construct systems where each step and module has a provable *worst-case* behavior.

Acknowledgments

We thank the anonymous reviewers as well as the organizers and participants of the New Security Paradigms Workshop 2005 for their input to the final version of this paper.

This research effort was partially funded by the National Science Foundation (NSF) under grants TC-0209163 and ITR-0205712 and by the Office of Naval Research (ONR) under agreement N00014-01-1-0854. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science foundation (NSF), the Office of Naval Research (ONR), or any other agency of the U.S. Government.

6. REFERENCES

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting Equality of Values in Programs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 1–11, San Diego, California, January 1988.

[2] A. W. Appel and K. J. Supowit. Generalization of the sethi-ullman algorithm for register allocation. *Software - Practice and Experience*, 17(6):417–421, 1987.

[3] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[4] CERT Coordination Center, Carnegie Mellon University, http://www.cert.org.

[5] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN 1982 Symposium on Compiler Construction (CC)*, pages 98–105, Boston, MA, June 1982.

[6] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, Martin, E. Hopkins, and P. W. Markstein. Register allocation via graph coloring. *Computer Languages*, 6(1):47–57, 1981.

[7] R. M. Cohen. The defensive Java Virtual Machine specification version 0.5. Technical report, Computational Logic, Inc., May 1997.

[8] S. A. Crosby and D. S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 2003 USENIX Symposium on Virtual Machines.* USENIX Association, 2003.

[9] P. Dadvar and K. Skadron. Potential Thermal Security Risks. In *21st IEEE SEMI-THERM Symposium.* IEEE, 2005.

[10] D. Dean and A. Stubblefield. Using Client Puzzles to Protect TLS. In *Proceedings of the 2001 USENIX Security Symposium.* USENIX Association, 2001.

[11] A. Gal, C. W. Probst, and M. Franz. An Applet performing a complexty-based Denial-of-Service attack on the verifier. Available at `http://nil.ics.uci.edu/exploit`.

[12] A. Gal, C. W. Probst, and M. Franz. A Denial of Service Attack on the Java Bytecode Verifier. Technical Report 03-23, University of California, Irvine, School of Information and Computer Science, 2003.

[13] A. Gal, C. W. Probst, and M. Franz. Proofing: An Efficient and Safe Alternative to Mobile-Code Verification. Technical Report 03-24, University of California, Irvine, School of Information and Computer Science, November 2003.

[14] A. Gal, C. W. Probst, and M. Franz. Integrated Java Bytecode Verification. In *Proceedings of the First International Workshop on Abstract Int erpretation of Object Oriented Languages*, January 2005.

[15] S. Garfinkel. Script for a king. HotWired Packet, `http://hotwired.lycos.com/packet/garfinkel/96/45/geek.html` and see `http://simson.vineyard.net/table.html` for the table attack., November 1996.

[16] L. George and A. W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[17] J. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[18] R. Joseph, D. Brooks, and M. Martonosi. Control Techniques to Eliminate Voltage Emergencies in High Performance Processors. In *HPCA '03: Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture (HPCA'03)*, page 79, Washington, DC, USA, 2003. IEEE Computer Society.

[19] S. Lelait, G. R. Gao, and C. Eisenbeis. A New Fast Algorithm for Optimal Register Allocation in Modulo Scheduled Loops. In K. Koskimies, editor, *Proceedings of the 7th International Conference on Compiler Construction (CC'98)*, volume 1383, pages 204–218, Lisbon, Portugal, March 28 - April 4 1998. Springer.

[20] X. Leroy. Java Bytecode Verification: Algorithms and Formalizations. *Journal of Automated Reasoning*, 30(3/4):235–269, 2003.

[21] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[22] V. Paxson. Bro: A System for Detecting Network Intruders in Real Time. *Proceedings of the 7th Security Symposium. (USENIX Association: Berkeley, CA)*, 1998.

[23] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[24] Redhat. Vulnerability in zlib library, Advisory ID: RHSA-2002:026-35, 2002.

[25] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbering and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 12–17, San Diego, California, January 1988.

[26] Sun Microsystems. The Java Hotspot Virtual Machine, 2002.

[27] Sun Microsystems. CDC: An Application Framework for Personal Mobile Devices, http://java.sun.com/products/cdc/, 2003.

[28] F. Yellin. Low level security in Java. In O'Reilly and Associates and Web Consortium (W3C), editors, *World Wide Web Journal: The Fourth International WWW Conference Proceedings*, pages 369–380. O'Reilly & Associates, Inc., 1995.