E unibus pluram: Massive-Scale Software Diversity as a Defense Mechanism

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

ABSTRACT

We contend that the time has come to revisit the idea of software diversity for defense purposes. Four fundamental paradigm shifts that have occurred in the past decade now make it viable to distribute a unique version of every program to every user. We outline a practical approach for providing compiler-generated software diversity on a massive scale. It is based on an "App Store" containing a diversification engine (a "multicompiler") that automatically generates a unique, but functionally identical version of every program each time that a downloader requests it. All the different versions of the same program behave in exactly the same way from the perspective of the end-user, but they implement their functionality in subtly different ways. As a result, any specific attack will succeed only on a small fraction of targets. An attacker would require a large number of different attacks and would have no way of knowing a priori which specific attack will succeed on which specific target. Hence, the cost to the attacker is raised dramatically.

Equally importantly, our approach makes it much more difficult for an attacker to generate attack vectors by way of reverse engineering of security patches. An attacker requires two pieces of information to extract a vulnerability from a bug fix: the version of the program that is vulnerable and the specific patch that fixes the vulnerability. In an environment in which software is diversified and every instance of every program is unique, we can set things up so that the attacker never obtains a matching pair of vulnerable program and its corresponding bug fix that could be used to identify the vulnerability. We propose a mechanism for incremental updating of diversified software that has this property.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.2.0 [**Software Engineering**]: General; D.3.4 [**Programming Languages**]: Processors—*Compilers*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Design, Reliability, Security.

NSPW'10, September 21–23, 2010, Concord, Massachusetts, USA.

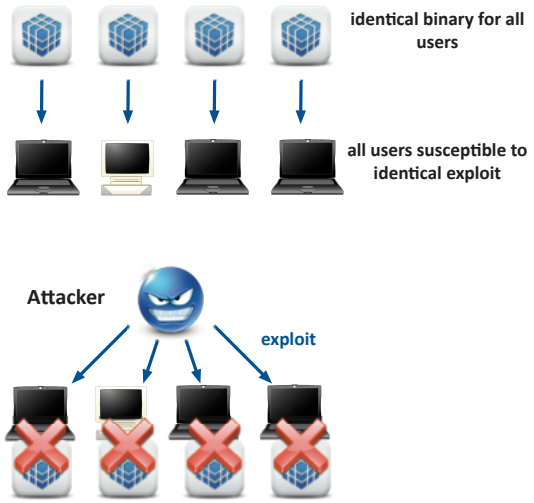

Figure 1: Software monoculture aids attackers.

Keywords

Compiler-generated software diversity, dynamic patching of software vulnerabilities, service computing architectures, software vulnerabilities, reverse engineering of security patches.

1. INTRODUCTION

Open networks and the computers on them are under constant attack from a variety of adversaries. Most of these attacks are enabled by software vulnerabilities, i.e., errors in operating systems, device drivers, shared libraries, and application programs that can be exploited to perform unauthorized operations on the computers running the software. Although considerable efforts have gone into finding and eliminating such errors, and although impressive advances have been made in doing so, the complexity of today's software systems is so great that a certain number of residual errors will probably always be present. The incidence of such errors tends to be proportional to the overall code size and decrease over time.

The existence of residual software errors becomes a significant threat when large numbers of computers are affected by the identical vulnerability at the same time. Unfortunately, this is the situation today. We currently live in a software monoculture—for some widely used software, the identical binary code is installed on millions of computers, and sometimes even hundreds of millions. This

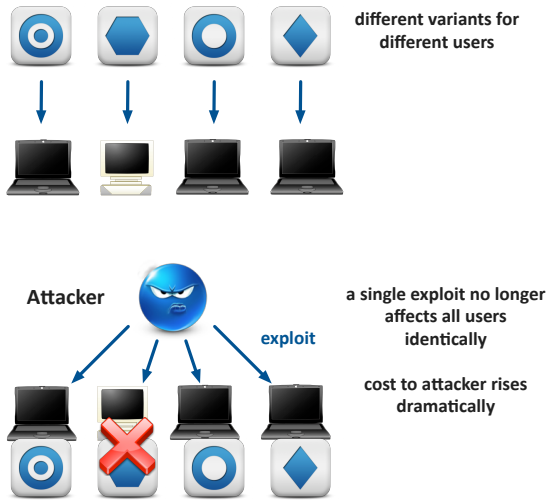

Figure 2: Software diversity lowers effectiveness of attack.

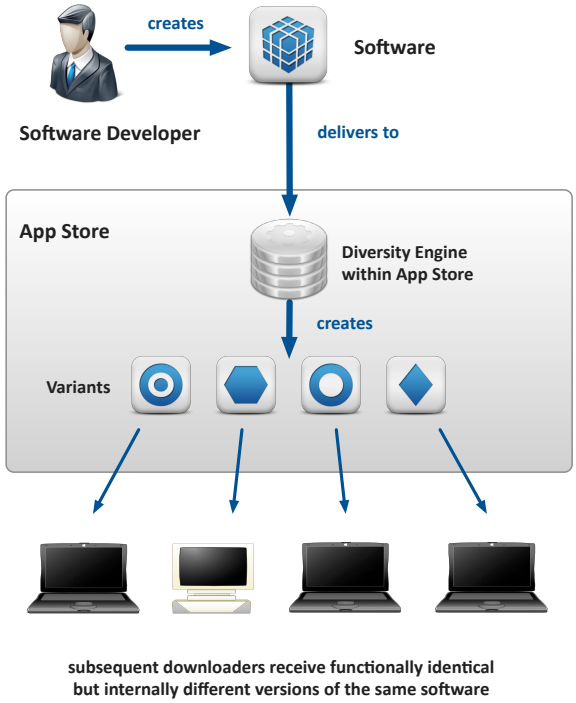

Figure 3: Diversification mechanism can be hidden entirely within an online software delivery system ("App Store") so that it becomes transparent to both code consumers and software developers.

makes it easy for an attacker (Figure 1), because the same attack vector is likely to succeed on a large number of targets [8, 19].

But what if these millions of computers were all running *different* versions of the software? That is, what if we could ensure that every computer runs a unique but functionally identical binary (Figure 2), so that a different attack vector is needed for different targets. All the different versions would behave in exactly the same way from the perspective of the end-user, but they would implement their functionality in subtly different ways. As a result, any specific attack would succeed only on a small fraction of systems and would no longer sweep through the whole internet. An attacker would require a large number of different attacks and would have no way of knowing a priori which specific attack should be directed at what specific target. Hence, the cost to the attacker would be raised dramatically.

The idea of using software diversity as a defense mechanism is not new, but it has never been realized in practice at any significant scale. Until quite recently, it would have been prohibitively expensive to create a unique version of every program for every client.

In this paper, we make a passionate argument that such massive-scale software diversity is now actually technically possible. We observe that this is enabled by four simultaneous paradigm shifts that are occurring just now. We present our blueprint for an architecture that provides such massive-scale software diversity. We elaborate on the problem of patching software that has been diversified. We present a list of interesting open research problems that appear in the context of massive-scale software diversity. We claim that massive-scale software diversity is a new security paradigm in itself. Finally, following a section on related work, we conclude the paper.

2. VULNERABILITIES, EXPLOITS AND A SOLUTION

A software vulnerability by itself is merely a hazard. In order to turn such a hazard into a successful attack, an attacker needs to find a successful exploit strategy. For example, the attacker may know of a vulnerability that enables a write beyond the end of a certain buffer on the stack. But in order to exploit this known vulnerability, the attacker needs to overwrite very specific locations on the stack with very specific values.

Operating system vendors now add elements of randomness to their systems, with the aim of making it more difficult for attackers to design a successful exploit. For example, the latest version of the Windows operating system now randomizes the starting address of the stack. Unfortunately, this has not stopped attackers from devising workarounds.

Designing a successful exploit for a known vulnerability is not trivial, but a dedicated attacker with ample resources is likely to succeed in eventually creating an attack. In today's world, the effort invested into designing such an exploit can be amortized by its wide applicability—since millions of users are running the identical vulnerable binary, just one successful exploit can affect all of them simultaneously.

It is this fundamental problem that massive-scale software diversity addresses. We advocate the introduction of automated code variance techniques that result in the binary code images delivered to subsequent code consumers being subtly different. This process can be embedded seamlessly into an online software delivery system, an "App Store" (Figure 3), and thereby be made entirely transparent to the code consumer (all programs derived from the same source in this manner have the identical functionality). The mechanism can even be set up so that adding software diversity poses no extra effort to the software programmer (a compiler automatically generates the different versions without any additional human intervention). But as a result of deploying massive-scale diversity, any specific exploit will work on only a relatively small number of targets.

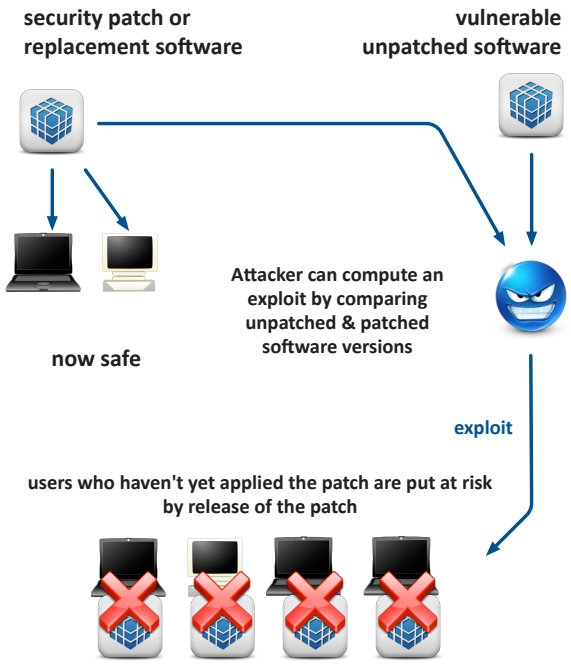

Figure 4: Existing situation: release of a software update exposes vulnerabilities.

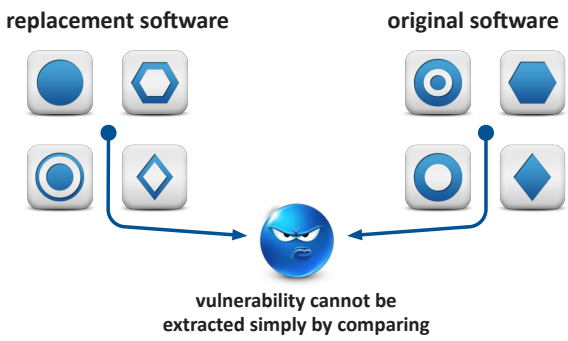

Figure 5: Solution #1: Software is updated by sending complete replacement version. Adversary cannot match an original version to its replacement and cannot extract a vulnerability.

Figure 6: Solution #2: Software is updated via custom patch. Patch is meaningless to adversary without the specific original version that it relates to.

3. DIVERSITY REMOVES REVERSE ENGINEERING VULNERABILITIES OF SOFTWARE PATCHES

Massive-scale software diversity removes another major problem of current software monoculture: the fact that releasing a patch for a discovered vulnerability alerts adversaries about the existence of the vulnerability. It is current best practice to fix software vulnerabilities as soon as possible after they are discovered. In the desktop space, this is usually achieved by sending a "patch" to the compromised host. Such a patch contains the delta between the original (vulnerable) program and a corrected new version. Since such fixes usually apply to only a small fraction of a program, it is more efficient to send just the patch rather than sending a whole corrected program.

In the mobile space, user-installable programs ("Apps") are currently updated by sending complete replacement versions rather applying an incremental patch, while the mobile operating system software itself is updated using patches. As Apps on mobile devices grow, it is probably only a matter of time until the App Store frameworks of the various mobile platforms will support replacing only part of an App (via a patch) rather than downloading a wholly new App each time.

A bug fix (in the form of either a patch or a replacement program) gives a potential adversary information that can be used to precisely identify the vulnerability being fixed in the new version (Figure 4). A significant proportion of software exploits today are generated from reverse engineering of error fixes. As a consequence, it is imperative that updates are applied as soon as they are available. The average time lag between availability of an update and its installation on a vulnerable target is often a good predictor for overall vulnerability.

In this context, Apps for mobile devices are actually potentially even more vulnerable than desktop applications. This is because Apps for mobile devices tend to evolve much more quickly than traditional desktop software. For example, for many Apps in the Android Marketplace, release cycles are expressed in days rather than months. The rapid software evolution cycle is more likely to push out software that is not as mature as it should be, i.e., containing a higher proportion of residual errors than necessary. And fixing these errors in subsequent releases will give an adversary a steady stream of hints as to the location of exploitable vulnerabilities.

Massive-scale software diversity makes it much more difficult for an attacker to generate attack vectors by way of reverse engineering of security patches (Figures 5 and 6). An attacker requires two pieces of information to extract a vulnerability from a bug fix: the version of the software that is vulnerable and the specific patch that fixes the vulnerability. In an environment in which software is diversified to an extreme degree and every instance of every piece of software is unique, we can set things up so that the attacker never obtains a matching pair of vulnerable software and its corresponding bug fix that could be used to identify the vulnerability. In Section 7 below, we outline a concrete mechanism that achieves this goal.

4. PARADIGM SHIFTS AS ENABLERS

Massive-scale software diversity is enabled by four fundamental paradigm shifts that have occurred almost simultaneously in the past few years. While each of these is remarkable in its own right, it is their fortuitous coincidence that is making software diversity truly scalable, affordable, and practical now. In the following, we

briefly describe each of these paradigm shifts, present evidence that indicates that the shift is real, and then discuss the consequences with respect to our vision.

4.1 Paradigm Shift One: Online Software Delivery

Traditional Approach:

Until quite recently, software was predominantly shipped "in boxes on a CD." Mass production of the CDs made it impractical to give every user a different version.

Paradigm Shift:

Distribution of a unique program version to each and every user becomes feasible when software is downloaded via the network rather than installed from a CD. We are just at the point when many programs are now installed only via the internet.

Evidence for Paradigm Shift:

The Firefox web browser has been downloaded via the internet more than one billion times.

New Approach Enabled By Paradigm Shift:

Rather than downloading the same binary to all users, it becomes possible to send each user a subtly different version with the exact same functionality. From the users' perspective, nothing at all has changed, but for an attacker, things have become a lot more difficult.

4.2 Paradigm Shift Two: Ultra-Reliable Compilers

Traditional Approach:

Not so long ago, the compiler itself was often the largest and most complex software program on any given system. Hence, it wasn't unreasonable to assume that the compiler itself might have errors. As a partial consequence, traditional software certification and testing has focused on the software binary that is the end-product of compilation. The idea of executing a binary program coming out of a compiler without any further testing of the binary itself was heresy not so very long ago.

Paradigm Shift:

Compilation is now a very predictable process. While almost all other software programs have grown in size and complexity, sometimes by orders of magnitude, compilers today are not orders of magnitude more complex than they were 20 years ago. Moreover, many existing compiler lines are very mature, having been *refined* rather than enlarged over sometimes decades. Without exaggeration, one can say that compilers are among the most reliable computer programs in existence. Even dynamic compilation is now routinely employed with extremely high reliability. Software errors that can be traced back to compiler errors are as good as unheard of. Just-in-time compilers and binary translators are now widespread, and as a result millions of users routinely execute binary code that comes straight out of a compiler, without any further testing prior to execution.

Evidence for Paradigm Shift:

Apple has transitioned millions of users from the PowerPC to the Intel architecture using a fully automated just-in-time compiler (binary translation engine) without any reported incidents. The reliability of these compilers is stunning, considering that they have been able to automatically translate programs of the size of the Microsoft Office suite fully unattended, without any testing of the resulting output, and on-the fly.

New Approach Enabled By Paradigm Shift:

Instead of testing and certifying a software binary, it should be sufficient to certify and test a *representative* binary coming out of a diversifying compiler ("multicompiler"). The purpose of this testing and certification would be strictly to find errors in the program, not errors in the compiler itself. We are assuming that we can build multicompilers that have the same reliability as current unicompilers, a property that could be verified by large-scale automated regression testing comparing different diversified program versions generated by a multicompiler against binaries produced by a unicompiler. If a difference in reliability were found, we should at least be able to quantify it.

4.3 Paradigm Shift Three: Cloud Computing

Traditional Approach:

In the past, it would have been impossible to set up the infrastructure that generates a unique version of each program for each user. The cost would have been prohibitive.

Paradigm Shift:

There are now remote computing platforms such as the Amazon Elastic Compute Cloud (EC2) that provide "cycles on demand" for a low fee. This makes it possible to scale almost instantaneously to even very large peak demands without any up-front investment.

Evidence for Paradigm Shift:

Many start-up companies are using cloud computing to rapidly scale up their operations. For example, Animoto is a company that lets customers upload images and music and automatically creates customized Web-based video presentations from them. In mid-April of 2008, Animoto was hit by a viral popularity surge through Facebook. After having 5,000 new customers on average per day, they suddenly had nearly 750,000 people sign up in three days. At the peak, almost 25,000 people tried Animoto in a single hour. Using cloud computing, they were able to successfully multiply their server capacity by a factor of almost 100 virtually instantaneously [6].

New Approach Enabled By Paradigm Shift:

Using cloud computing, the cost per unique version of a program is essentially constant, no matter whether we are generating 1000 versions per day or 10 Million versions per day, and we can react to changing demand almost instantaneously.

4.4 Paradigm Shift Four: "Good Enough" Performance

Traditional Approach:

Most of our computing past has been dominated by Moore's Law: computers were always getting faster, software kept growing more complex, and users were willing to upgrade to the newest and latest hardware to be able to run the latest software with the newest features.

Paradigm Shift:

Suddenly, performance in some domains has become "good enough." This does not apply to every domain; for example, games are a no-

table exception. But for many traditional desktop computing categories, just at the time that Moore's Law apparently has hit a wall at which hardware manufacturers find it increasingly difficult to further raise clock frequencies, users apparently have mostly decided that they now have sufficient computing power anyway.

Evidence for Paradigm Shift:

Microsoft has had a difficult time persuading users to upgrade from Windows XP, and fewer and fewer users see the utility of upgrading to the latest version of Microsoft Office. At the same time, a wide range of "netbooks" has appeared in the marketplace that offers noticeably less (sometimes by as much as 50%) processing performance than established hardware, and yet these inexpensive weakly-powered computers are proving to be extremely popular, because they are "good enough."

New Approach Enabled By Paradigm Shift:

Because software performance is now mostly "good enough," users are likely to accept a small performance penalty if it gives them added security. So even if a multicompiler were to create program versions that are less efficient by a small degree, say 5% to 15% less performance than the optimal version created by a unicompiler, this no longer automatically dooms the prospect of massive-scale software diversity becoming a success. Users no longer care so much about performance and may be willing to accept additional security as a welcome trade-off in return for a slight runtime cost.

5. RUNTIME COST OF ALTERNATIVE CODE PATHS

So the next question to ask is: will there actually be a runtime overhead, and if yes, why and how much? Since no large-scale system such as the one we envision has been built, we cannot give a definitive answer to this question, but having significant expertise and implementation experience in the area of compiler construction, we can give an educated estimate.

Today's unicompilers are focused on finding the "best" of several possible binary implementations of any source construct. There are usually many alternative paths to choose from, and compilers use heuristics to choose the one that is most likely to provide the best runtime performance. Instead of choosing the "best" of the alternative paths, a multicompiler would give successive users different code paths. Some of the alternative paths will not be as optimized as the "best" one.

The potential performance loss comes from the difference between the "best" path and an alternative path chosen by the multicompiler for the sake of implementation diversity. In many cases, there will be no performance difference at all. Very often, there are many alternative paths that have exactly the same cost. The main difference between a unicompiler and a multicompiler in these cases will be that the unicompiler always chooses the exact same path in a reproducible manner when confronted with such a choice, whereas the multicompiler will attempt to randomize among the equivalent paths.

In many cases, there will be sufficiently many alternative paths that all have the "best" runtime behavior, so the multicompiler will never have to choose a path that leads to a performance degradation. But even if a sub-optimal path needs to be chosen now and then for implementation diversity, we don't expect any significant performance loss. When we study compiler optimization results from academic conferences such as PLDI (Programming Language Design and Implementation) and CGO (Code Generation and Optimization), we find that the incremental benefit of even quite sophisticated optimizations is usually surprisingly small. In the field of compiler construction, a speedup of 3% to 5% is often already considered a significant publishable result. By some measure, this is a sign of the maturity of the field. But it also means that the difference between the "best" path and a "sub-optimal" alternative path is going to be of similar magnitude. After studying the literature on performance variations resulting from existing compiler optimizations, we expect that a slowdown of 5% is probably the maximum runtime cost that would arise out deploying "standard" compilation mechanisms to achieve code variability.

Furthermore, hardware evolution keeps diminishing the performance differential between the "best" and a "sub-optimal" code path. For example, consider the pervasive use of caches in modern processors. In conjuction with out-of-order instruction execution, this has significantly reduced the importance of a good register allocator. In many cases today, it no longer matters from a performance perspective whether a value is in a register or in the cache. But from the perspective of software diversity as seen by an attacker, this is a fundamental difference that requires a completely different plan of attack.

One could of course also consider variation mechanisms that lie outside of the scope of existing compilers precisely because they clearly degrade performance. In certain high-assurance contexts, one may want to explicitly sacrifice execution performance for even greater code variability.

6. COST OF COMPILERS IN THE CLOUD

Cloud computing has seen explosive growth over the very recent past and is now available from many commercial providers. For example, Amazon's Elastic Compute Cloud (EC2) is a service that provides remote rental of dedicated compute servers over the network. The service offers a selection of standardized computer configurations to choose from, so that performance is predictable, and is billed per instance-hour consumed.

Running a compiler is mostly a compute-intensive medium throughput process. In order to estimate what it would cost to provide such a service "in the cloud," we measured the time required to compile the open-source Firefox browser on a modern server dedicated exclusively to this task. We chose Firefox as our model program because on one hand we are contributors to the browser and have its complete source tree readily available, and because on the other hand, most people are familiar with Firefox no matter what platform they use in their day-to-day computing tasks. Firefox is a very substantial program; it has about 30 million lines of code. As a matter of comparison, Linux 2.6.33 (released February 2010) contains about 13 million lines of code, and Windows 7 reportedly consists of approximately 50 million lines of code.

The time to compile Firefox on a dedicated server is about 30 minutes. Hence, in order to create 1000 diversified instances of Firefox, we would need about 500 compute hours from a cloud computing provider.

The server we used to compile Firefox is roughly equivalent to Amazon's "High-CPU Medium Instance" that can be rented by the hour via the EC2 service. Using Amazon's cost estimate web page, we find that renting such a "High-CPU Medium Instance" server for a month (730 hours) with 100 GB of upload bandwidth and 200 GB of download bandwidth would cost $131.60 at early 2010 pricing levels. This translates into a price of 9 cents per build for an application the size of Firefox, a price that could easily be absorbed into the retail cost of a commercial product and that most users might be more than willing to pay to voluntarily obtain their own custom "diversified" version of an otherwise "free" product.

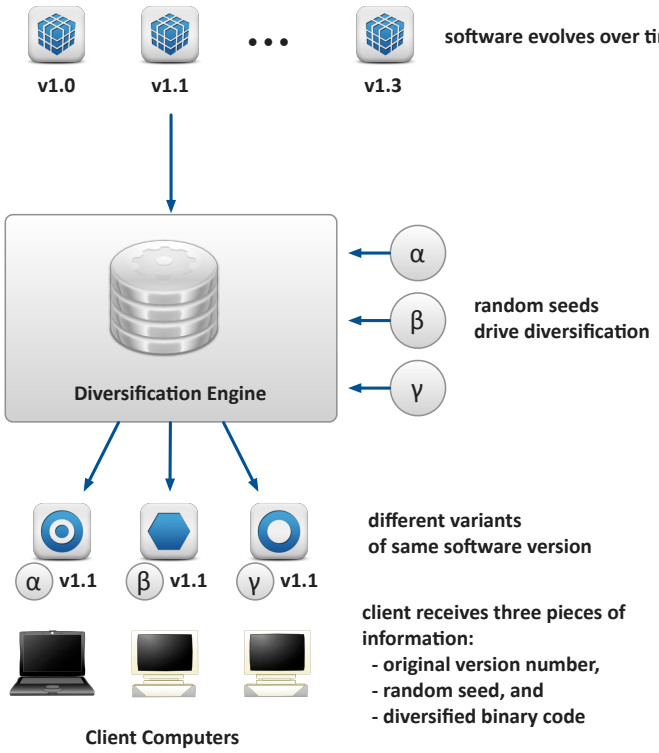

Figure 7: Initial generation and delivery of software variants.

7. SOFTWARE UPDATES AND PATCHES

An interesting problem arises from the necessity of updating software that has been diversified. In Section 3 above, we already mentioned the current practice of providing "delta only" software updates through a "patch" that modifies the binary image on the client computer from the old version to the new version. This approach is no longer so straightforward when each client computer is running a different binary. While a straightforward solution would consist of simply having each client download an entirely new diversified version of the updated software, this would often result in very voluminous downloads for only relatively small actual changes.

On the other hand, there must be no mechanism by which an attacker could determine which specific version of a binary is running on a client. Hence any update mechanism that transfers only the "delta" between versions must be driven by the client, in that the client communicates to the update mechanism which specific paths were chosen in the compilation of its software version, and the update mechanism then crafts an update patch specific to the particular client. Note that patch construction can yet again be farmed out to a cloud computing mechanism and is thereby almost infinitely scalable.

As a possible solution, we envisage a multicompiler that chooses a different random seed for each version it generates. The random seed drives a random number generator that in turn drives selection of code paths during compilation. The eventual delivery to the client has three components: the random seed, a unique vID identifying the version of the source program that was used for generating the diversified software binary, and the diversified binary itself (Figure 7).

When the client later requests an update, it returns the random seed and the source program vID to the update mechanism. The update mechanism contains a multicompiler that is run both on the new version (to be installed on the client) and on the old version currently installed on the client (identified by the vID), using the random seed to drive the diversification. This results in two Apps on the updater, a new version (to be the end result of the update on the client) and an old version (the identical App that is currently installed on the client). The updater can then compute a patch by comparing these two versions (Figure 8).

One could imagine even smarter ways of providing updates. For example, one could decompose the original software into "tiles" along procedure boundaries or similar compiler-related constructs. When an update occurs, a reachability analysis in the compiler would compute which of the tiles are affected by the change. When computing the patch, the multicompiler would then only need to operate on the tiles that are affected by the change, often greatly reducing the effort required for the update. The update mechanism could also make a trade-off decision between sending a full new version (at higher bandwidth costs) or computing a patch (at higher compute costs).

Among the more advanced open problems is the question of how one would implement *remote attestation* of software metrics in the absence of a fixed binary from which a hash can be computed. There are scenarios in which a client computer might want to "prove" to a remote server that it is running a specific "well known version" of a program that has not been tampered with. For instance, access to a video streaming service might require the client to "prove" that it is incapable of recording the stream. New research will be needed to answer the question of how to do this in a world in which every instance of the client software is unique.

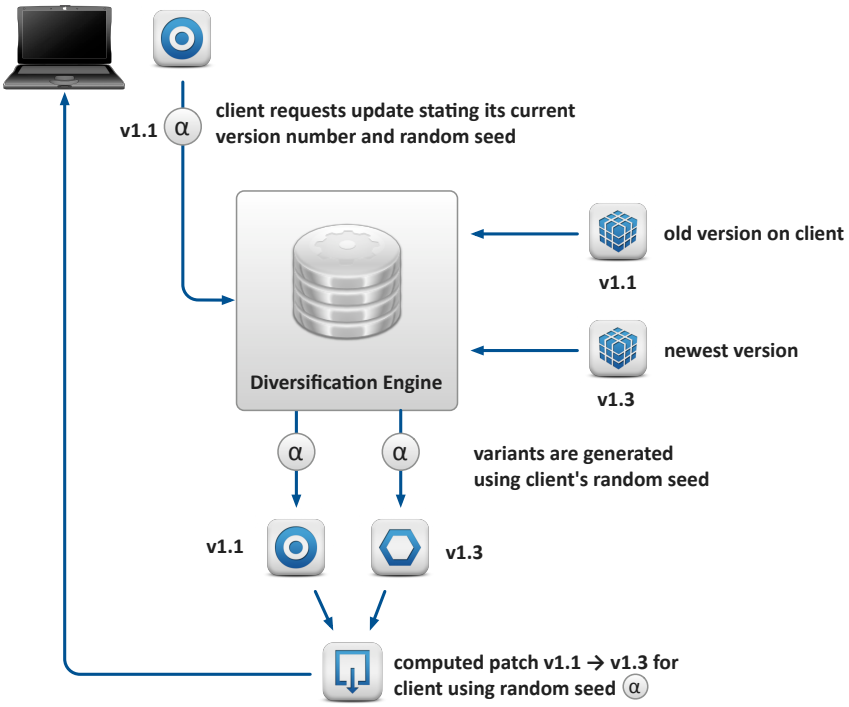

Figure 8: Generation and delivery of patches on demand.

8. TRUSTING THE DELIVERY SYSTEM

Users might be hesitant to download software without being able to determine whether a binary received over the network is legitimate. In the current "single version" practice, it is possible (but not customary outside specific high-assurance domains) to compute a fingerprint or checksum of a downloaded binary and compare it to the value of a "known good version." This capability disappears when every downloaded binary is unique.

One way of establishing trust in code transmitted over a network has in the past been the *verification* of the code itself prior to execution. For example, the Java bytecode format contains intrinsic provisions for the client to check the type safety and referential integrity of a program prior to its execution. This approach is limited to programs written in certain type-safe languages, such as Java and C#.

Interestingly, all of today's existing "App Store" platforms have more or less abandoned the approach of code verification on the target, even in such cases where the Apps have actually been developed in type-safe languages and on-device verification would in principle be possible. Instead, code verification, if used at all, has been moved "upwards" in the delivery pipeline (Figure 9). For example, on the Android platform, Apps are developed in Java using the standard Java development toolchain. But then they are converted off-line into a *type-unsafe* execution format for the register-based *Dalvik* virtual machine, which can be executed more efficiently than Java bytecode and is not subject to the Java licensing restrictions.

As a consequence, in most of the existing mobile device platforms, the delivery system *must be trusted*. In the Apple iPhone/iPad/iTunes ecosystem, developers deliver binaries in the ARM processor's native instruction format to the App Store, while in the Android ecosystem, developers usually deliver binaries in the Dalvik virtual machine's native bytecode format (.dx)—although apparently also some native ARM applications exist. The whole path between the developer and the eventual mobile device must be protected against tampering, which includes the necessity of having to protect the applications as they are hosted in the App Store itself.

The software delivery system we have sketched out here follows this established "App Store" architecture and adds the variant-generating "multicompiler" as an additional step in the path from code producer to code consumer. As in the existing App Store delivery systems, our approach requires that the generated versions queued up for delivery are protected against tampering, and it requires that the delivery handshake to the target is secured (via some cryptographic mechanism). Our approach is applicable both to executable machine code as well as to the bytecodes of virtual machines such as Dalvik.

Another potential issue is the reliability of the code variability generator itself. In Section 4.2 we talked of "ultra-reliable" compilers as a paradigm shift, and compilers have indeed become extremely reliable. There are many deployment scenarios today where users routinely execute code coming out of a compiler without any further testing done on the end product of compilation. On one hand, language runtimes such as the Java Virtual Machine and Microsoft's Dot Net Common Language Runtime have provided just-in-time compilation for a decade now. On the other hand, binary translators have been deployed "under the hood" in products such as Transmeta's family of processors (mapping x86 to a custom VLIW architecture) and Apple's Rosetta engine (mapping PowerPC code to x86). No matter how successful and trouble-free these compilers have proven to be, it may yet present a challenge to persuade users to accept unsupervised automatic compilation without subsequent testing as an integral part of the default software delivery pipeline.

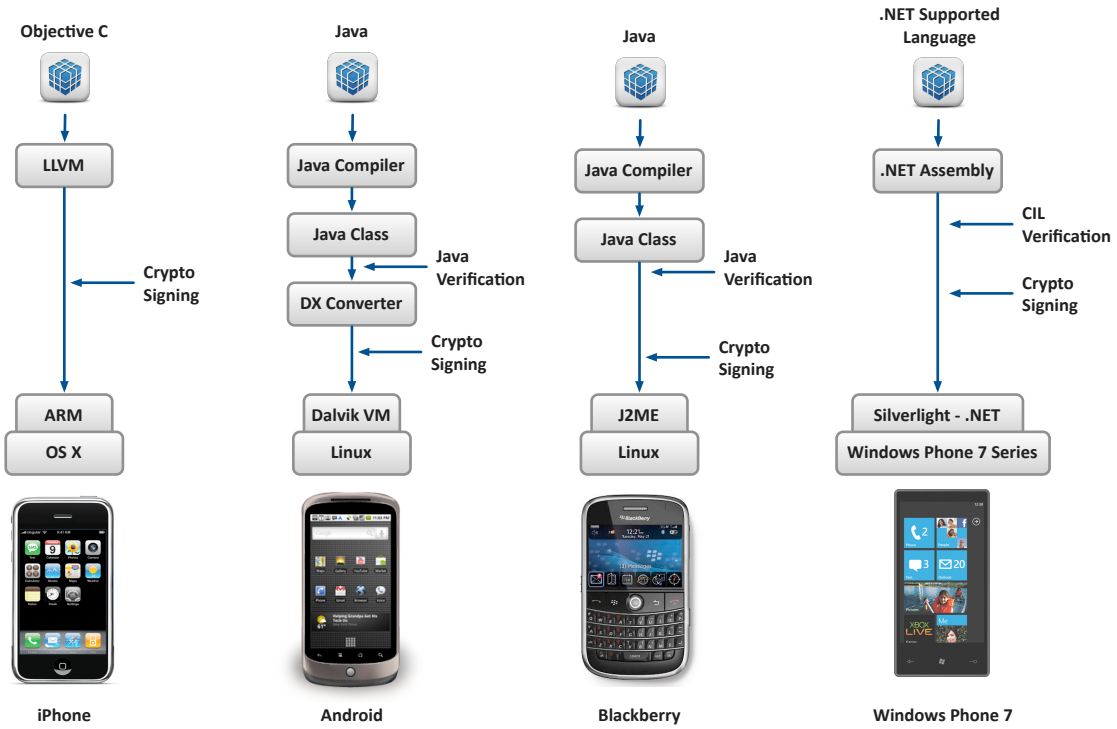

Figure 9: Current code-signing based approaches.

9. OPEN RESEARCH PROBLEMS

Massive-scale software diversity does not yet exist. As a result, almost all questions regarding metrics, i.e., what are the costs versus the benefits of certain design choices, are completely open at this time and cannot really be answered without building an actual working system and experimenting with it. But even when it comes to the overall architecture and mechanisms, there are surprisingly many alternatives.

We present some examples of interesting research questions. Most probably, these questions only scratch the surface of the design space and even more interesting ones will emerge as someone actually implements a massive-scale software diversity system.

9.1 Generating Alternative Paths

Instead of choosing just a single "best" code path for a source construct being compiled, a multicompiler chooses among many alternative paths. At first glance, this is not very different from the activity of a traditional compiler, except that the alternatives are preserved rather than discarded in favor of the "best" one. But when digging deeper, many intriguing questions emerge. For example:

- How do you choose among alternative paths in such a way that the process is reproducible when needed (e.g., for generating patches), but simultaneously doesn't give any advantage to an attacker? There are pobably even better solutions than the idea of a random seed that drives a random number generator, as outlined above in Section 7.

- When there are a great many alternative paths at any choice point, can we limit the choice to only those paths that won't incur a performance loss? How many different choices do you need at minimum to achieve the desired variability?

- Some code paths are correlated. For example, choosing a certain register allocation at some choice point constrains other code paths "downstream." How do we navigate these dependencies to ensure that sufficiently many versions can ultimately be generated?

- Is there perhaps a trade-off between maximizing inter-version code variability (by making the path chooser take large strides) and prematurely exhausting the reachable path space? Does that suggest we should have an a priori estimate of the anticipated number of unique versions to be ultimately generated from any source program so that we can choose the correct inter-version path distance?

9.2 Variability Techniques Beyond Current Compilers

There are probably diversity-enhancing techniques outside of the scope of existing compilers that could further increase the variability to an attacker without changing the functionality for the end-user. This would include mechanisms that incur more substantial runtime costs than merely not choosing the "best" path. One could probably borrow ideas from existing research on code obfuscation [4].

Among the techniques employed by code obfuscators is control-flow obfuscation, i.e., modifying the control flow of a program by re-distributing actions across basic blocks without changing actual program behavior. Unlike code obfuscators, we are less concerned about algorithm recapture and more focused on enhancing instruction-level code variability. As a result, a multicompiler will probably focus less on code obfuscation techniques that insert superfluous control-flow paths into a program, but it may readily employ code re-factoring strategies.

We envisage a scenario in which large-scale software diversification will eventually be applied to all software, including system libraries and large parts of operating systems. Not only will this mitigate attacks that are exploiting errors in the libraries and operating systems themselves, it will also help to defeat "arc injection" attacks that use existing library code sequences ("gadgets") as stepping stones [18]. To this effect, code variability techniques will need to be engineered to prevent a canon of identical "gadgets" to exist across many diversified versions of the same library.

9.3 Systemic Properties

Assuming that a functioning multicompiler can be implemented, the research community could then leverage the vast collections of historical vulnerabilities and exploits to determine some more "systemic" unknown parameters of the proposed approach. Some example open questions of this kind include the following:

- Is the concept more defeatable if the variation engine is predictable? Is this approach more like cryptography, which depends on keeping a key private, even if the algorithm is known, or will knowledge of the algorithm reduce the effectiveness of the technique?
- Even if a flaw cannot be exploited to fully compromise a system, what does this technique do to avoid simply taking a system down (crashing it) by corrupting the stack? That is to say, can we systematically choose variations that will ensure *survivability* of the majority of versions, beyond merely guarding against takeover by an adversary.
- Given how many possible vulnerabilities are likely to exist, what are the odds that there will be enough possibilities for diversity to cover all possible vulnerabilities? How much diversity is possible in comparison to the amount of code running, and the number of vulnerabilities or flaws in that code?

Clearly, it would be valuable to know the answers to these questions.

10. CLAIM OF A NEW PARADIGM

The author hopes to have convinced the reader not only that massive-scale software diversity is now within practical reach, but also that it will usher in a new paradigm of software security. Many, if not most, of the assumptions and models underlying current computer security threats are a direct result of the existing software monoculture. Computer viruses and worms, root kits, botnets, ... — the root cause for the existence of all these troubles is the fact that too many computers run the identical software binaries. Adopting a strategy of uniqueness of every single program on every single host *at internet scale* is something fundamentally new, a paradigm shift.

11. RELATED WORK

The idea of using diversity to improve robustness has a long history in the fault tolerance community. The basic idea has been to generate multiple independent solutions to a problem (e.g., multiple versions of a software program, developed by independent teams in independent locations using even different programming languages), with the hope that they will fail independently. The expectation is then that at any given point in time, a majority of the versions will be functioning correctly [14, 1]. An abundance of evidence suggests that such n-version development techniques are more reliable, and more cost-effective, than producing one "good" version, especially in situations where the cost of failure is high [9]. This is in spite of the fact that many n-version software systems in practice exhibit a surprising amount of coincident failures of multiple supposedly independent program versions [12, 5].

More recently, along with a rising awareness of the threat posed by an increasingly severe computer monoculture, diversity has also been proposed as a means for improving software security. Most of the past approaches have been based on some form of obfuscation [3, 13]. Some research ideas on operating system randomization [2, 20] have since found their way into commercial operating systems. Other suggested obfuscation techniques have included load-time binary transformation [11] or even "private machine architectures" based on virtual machines [10].

Unlike these approaches, massive-scale software diversity as presented here is not primarily focused on obfuscation. Instead, it exploits the randomness that is already inherent in compilation (in many cases, alternative paths are truly equivalent and the choice made by current unicompilers, although consistent, is algorithmically arbitrary). In this respect, the scope of this technique also goes several orders of magnitude beyond earlier work by Forrest et al. [7] that pioneered the idea of compiler-guided code variance. Thirteen years of technical innovation since that earlier work and the advent of cloud computing have changed the landscape fundamentally. It is now perfectly feasible to create a unique custom version of every program in existence for every user that wants one, and the cost estimate of 9 cents for a significant program that we gave in Section 6 is entirely realistic. Moreover, each of these unique versions is created by a full start-to-finish compilation process of the whole program, rather than merely parametrizing some part or performing only "peephole" optimizations, so that every conceivable valid permutation of the whole program is a possible option for the code variation generator.

Finally, massive-scale software diversity is related to the author's own earlier work on combining software diversity with parallelism and checkpointing [17, 15, 16]. The earlier approach was to run several slightly different versions of the same program in lockstep on a multicore processor and monitor for differences in behavior. The main emphasis was on the monitoring and control layer: because the versions operate in lockstep but behave like just a single program, system events such as user input, file accesses and signals need to be virtualized and dispatched to the various versions at the same logical (as opposed to temporal) point. Further, the need to run the versions in lockstep with relatively small skew limits the variations that are possible between versions. The approach introduced here has no such constraints, because each version is run independently of all the others. As a consequence, far more ambitious code variations can be employed.

12. SUMMARY AND CONCLUSION

Adopting massive-scale software diversity will have a dramatic impact on the way that software is distributed and is likely to change many of the assumptions and models underlying current threats to deployed software. When every software binary is unique, it becomes much less likely that a single attack will affect large numbers of targets simultaneously. Hence, the impact of phenomena such as "viruses" and "worms" will be greatly reduced.

More subtly, the distribution of unique binaries also has the effect that adversaries can no longer simply analyze their own copies of any given piece of software to find exploitable vulnerabilities, because any vulnerabilities they may find will no longer automatically translate to all other instances of the same software. Hence, even *directed* attacks against *specific* targets running some unique

version of some software will become much more difficult, as long as the attacker has no way of determining which specific binary is present on what target.

Finally, massive-scale software diversity makes it much more difficult for an attacker to generate attack vectors by way of reverse engineering of software updates. We have outlined a mechanism for intelligently updating software that has been diversified at internet scale.

Without doubt, the new paradigm of massive-scale software diversity will change many of the existing approaches to software security. It will make the digital domain safer. It will also be a great canvas for researchers, because it presents many challenging open problems.

13. ACKNOWLEDGEMENT

The artistically challenged author wants to thank Michael Bebenita for turning his rough pencil sketches into the stunning computer-drawn figures you see in the paper. Thanks also go to one of the anonymous reviewers for pointing out some additional "systemic" research questions that have now been added to Section 9.3.

Parts of this work were supported by the United States National Science Foundation (NSF) under Grants No. CNS-0905684 and CNS-0627747. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the National Science Foundation.

14. REFERENCES

[1] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *IEEE COMPSAC 77*, pages 149–155, 1977.

[2] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, Dec. 2002.

[3] F. Cohen. Operating system protection through program evolution. *Computers and Security*, 12(6):565–584, Oct. 1993.

[4] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison Wesley, 2009.

[5] D. E. Eckhardt, A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk, and J. J. P. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Transactions on Software Engineering*, 17(7):692–702, 1991.

[6] M. Fitzgerald. Cloud computing: So you don't have to stand still. *New York Times, May 25th*, 2008.

[7] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 67–72, 1997.

[8] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. P. Pfleeger, J. S. Quartermain, and B. Schneier. Cyberinsecurity: The cost of monopoly: How the dominance of Microsoft's products poses a risk to security. Technical report, Computer and Communications Industry Association, 2003.

[9] L. Hatton. N-version design versus one good version. *IEEE Software*, 14(6):71–76, 1997.

[10] D. A. Holland, A. T. Lim, and M. I. Seltzer. An architecture a day keeps the hacker away. *SIGARCH Computer Architecture News*, 33(1):34–41, 2005.

[11] J. E. Just and M. Cornwell. Review and analysis of synthetic diversity for breaking monocultures. In *2004 ACM Workshop on Rapid Malcode (WORM '04)*, pages 23–32, 2004.

[12] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, 1986.

[13] C. Pu, A. Black, C. Cowan, and J. Walpole. A specialization toolkit to increase the diversity of operating systems. In *ICMAS Workshop on Immunity-Based Systems, Nara, Japan*, Dec. 1996.

[14] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1:220–232, 1975.

[15] B. Salamat, A. Gal, and M. Franz. Reverse stack execution in a multi-variant execution environment. In *2008 Workshop on Compiler and Architectural Techniques for Application Reliability and Security (CATARS'08)*, June 2008.

[16] B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. Franz. Multi-variant program execution: Using multi-core systems to defuse buffer-overflow vulnerabilities. In *2008 International Workshop on Multi-Core Computing Systems (MuCoCoS 2008)*, March 2008.

[17] B. Salamat, T. Jackson, A. Gal, and M. Franz. Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Eurosys 2009*, April 2009.

[18] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007*, pages 552–61. ACM Press, Oct. 2007.

[19] M. Stamp. Risks of monoculture. *Communications of the ACM*, 47(3):120, 2004.

[20] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 260–269, 2003.