The polyglot computer[*]

Daniel Medeiros Nunes de Castro
Department of Computer Science
University of Calgary
2500 University Drive N.W.
Calgary, AB, Canada T2N 1N4
dmncastr@ucalgary.ca

TR 2013-1048-16, October 2013

Abstract

Performing security verifications on a compromised system can give a false sense of security. If compromised, a computer system can return false results, thus "deceiving" the verification process. Our motivation for this work is straightforward: Computers should not be trusted, at least not when they are attesting their own integrity.

In our project Babel, this problem is addressed by, quite literally, thinking outside the box. Babel introduces an architecture where the user's computer is unable to execute any program by itself and depends on an external entity to execute any application. Taking into consideration the advances in computer network and cloud computing, we move the verification process to outside the physical limits of the computer.

Babel can be mistaken for yet another instance of extant approaches. In this paper, we revisit the Babel architecture with the twofold intention of clarifying what Babel is and showing how Babel differs from previous work.

1 Introduction

Popular mechanisms and methods for evaluating the integrity of a computer system rely on the information given by the very same computer we want to evaluate. That computer is used for both collecting information and performing the evaluation itself.

In this project called *Babel*, we argue that this popular solution starts from a false premise, i.e., we want to believe that the computer will always tell us the truth. When, in fact, it is very likely that a compromised computer would

[*]This paper was presented in the Ph.D. talks (NSPHD) of NSPW 2013.

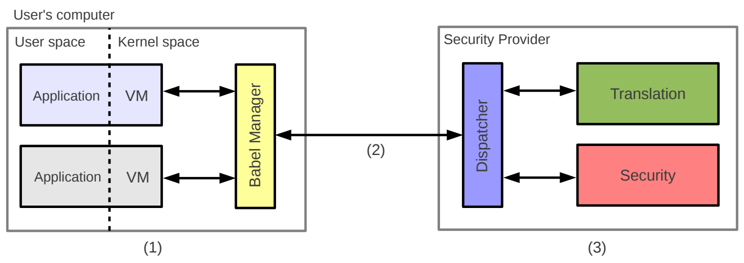


Figure 1: Babel architecture

produce a result based on false information, or that the evaluation process itself would be directly affected or attacked.

With Babel, we address this problem by taking a rather radical approach. The computer becomes, by itself, incapable of executing any program without the intervention of a trusted third party. This third party will ensure that the code is secure for execution. A relationship in which a party strongly depends on another for any decision is known in psychology as “co-dependency” [29, p.8]. In Babel, we introduce the idea of co-dependency as a means to computer security.

The aforementioned trusted third party, which we call a *security provider*, has two main responsibilities. First, the security provider must help the computer to understand and execute the program, which is initially incompatible with the computer. Second, and maybe more important, the security provider must also perform security checks on any and all code that will be executed, *before* that code is in fact executed. That includes dynamically generated code and even code that somehow happened to be injected into memory.

Babel was designed in the form of three elements, represented in Figure 1: (1) a local component located in the user's computer; (2) a remote component located in the security provider's environment; and (3) the communication channel used to connect the user's computer to the security provider. The remainder of this section will discuss the Babel architecture and each of these elements and discuss some of the advantages and disadvantages of using Babel.

1.1 Babel architecture

The first element in the Babel architecture is the *Babel client*. The client ideally consists of a specially crafted operating system that runs on the user machine. An ideal operating system would have a minimal trusted base. Our suggested approach consists of a micro-kernel based operating system, such as Exokernel [12]. A small kernel suggests that there may be less opportunities for vulnerabilities in the kernel code, or at least that thoroughly auditing the code may be possible.

An alternative and non-ideal implementation of Babel would consist of having the Babel client as a module in a stock operating system. (In fact, in the current stage of our work, we have chosen the latter approach for the implementation of a proof-of-concept.)

To implement the aforementioned co-dependency, all programs run on application virtual machines, one VM for each program, that are incompatible to the program itself. To actually perform the code execution, the VM needs to communicate to a security provider in order to obtain a translated version of the program it is executing. This translation must happen incrementally, in order to ensure that the actual code being executed is translated, independently of how the code is represented in the file. An incremental translation also ensures that any code injected by an adversary also needs to be translated and, thus, checked for malicious features prior to its execution.

The *Babel Manager* is the component on the client side that is responsible for the above tasks. The Babel Manager instantiates a VM for each user application being executed in execution, and is also responsible for the communication with security providers.

Each instance of a VM is unique in Babel. This uniqueness lies in each VM using a different "language". In the context of Babel, a language is defined by the set of characteristics that describe a VM, such as instruction set, register set (if any registers at all), memory organization and other characteristics. With a different VM for each process, we include program diversity as an integral part of Babel.

However, in Babel, the importance of program diversity is not limited to helping to enforce co-dependency; it also adds a layer of protection against code-injection [16]. In our current proof-of-concept, described in [3], we have used a variation of ISR [6] to provide low level diversity; our languages differ one from another simply by the instruction set[1]. A random key is selected by the security provider and sent to the client to encrypt instruction opcodes during the VM instantiation. In order to make it harder for an attacker to recover the key, the key is discarded from the client's memory after creation of the VM.

By choosing a different language, or in our proof-of-concept by encrypting the instruction set, we essentially make the program we want to execute incompatible with the VM that will execute it. During execution, a Babel-enabled VM needs to send each and every instruction to a security provider. The security provider then translates the instructions to the language used by that specific VM on the client, and returns it so the VM can execute it.

The second element of the Babel architecture is the remote component. In the most basic implementation, this component can be implemented as a service running on a computer in the network. As such, we refer to the remote component as the *Babel server*. Some of the scenarios in which we can deploy the Babel server are described in [3], and it includes having one client communicating with multiple servers for improved security. The Babel server itself consists of three main components: a dispatcher, that controls the connection to the clients and also decides what to do with each received message; a translator, that, as the name suggests, translates the code; and a security module, that is responsible for performing security checks.

[1]See Section 2.1 for a brief discussion on diversity.

The translator and the security module might, optionally, also perform some modification on the code. Some of the possible modifications include code optimization and the injection of security checks that might need to be performed on the client side.

The third element is the Babel communication channel. The channel is assumed to be secure, thus it should ideally implement authentication and be encrypted to avoid eavesdropping. Our protocol includes messages for translation and also control messages that perform special operations on the client, for example, triggering an error. A complete description of the Babel communication protocol is also given in [3].

2 Dissecting Babel

When designing Babel, we based some of the design decisions on other existing projects. However, rather than making Babel a different instance of such projects or ideas, we used them as building blocks. Nevertheless, such building blocks should not be confused with the final product that is Babel, as a single brick or even any room cannot be mistaken by the house to which they belong.

In this section, we dissect Babel and discuss each of those parts. Particularly, we discuss the aspects of diversity and virtualization on the client side; and the code interpretation.

2.1 Diversity and ISR

As we previously discussed, program diversity plays an important role in Babel. First introduced[2] by Forrest et al. [13], the basic idea of program diversity for security is that having different versions of the same program results in a more robust overall system. Even if a program contains some vulnerability, a single exploit would not affect all the different versions of that program.

Yet, program diversity is used in Babel mainly as a tool to enforce co-dependency and not as a final goal. It is, after all, the co-dependency that drives Babel to perform security checks completely isolated from the suspicious program.

Instruction set randomization (ISR) [16] is an approach for program diversity that focuses on defending against code injection. ISR consists of having a different instruction set for each instance of a program. This is achieved by randomly choosing a key and encrypting the instruction set of a program in memory.

In our proof-of-concept, as we mentioned in Section 1.1, we have implemented program diversity by performing ISR for each application in execution.

While we currently focus on instruction level diversity, ISR is not the only way program diversity can be implemented in Babel. We envision having completely different languages. Instead of simply encrypting or encoding the in-

[2]Forrest et al. in fact acknowledges that diversity itself was first suggested by [11]. However, Forrest et al. introduced practical approaches for program diversity.

struction set, we could have different instruction sets per program, different memory organization approaches, even different types of virtual machines to execute a program (e.g., stack- versus register-based machines).

2.2 Virtualization

There are usually two classic forms of virtualization for computers: *system* and *application*[3] virtual machines [27]. A system VM consists of virtualizing an entire computer, thus requiring even the installation of an entire operating system. Examples of system VMs include VMWare[4] and Xen [5]. An application VM, on the other hand, consists of only a layer of abstraction between the program and the operating system. An application VM is usually instantiated for each program in execution and the VM itself is considered a process in the operating system. Examples of application VMs include the Java VM [20] and Inferno's Dis [24].

The use of virtualization techniques in Babel aims particularly at enabling program diversity. However, the virtualization obviously brings other advantages, such as isolation between programs.

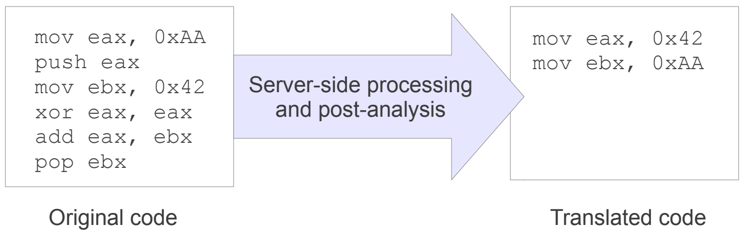
An alternative approach to Babel that has been often suggested consists of having our security provider running locally, in a different virtual machine, instead of a different and remotely located computer. This would certainly reduce the latency and the security checks would still be performed in a somewhat different machine, thus isolating program execution from security checks. In fact, there has been work that attempt to achieve a similar result (e.g., [10,14]). However, it is not really clear whether VMs are really isolated from one another. Despite that, the massive movement towards using VMs for security have driven VMs to become increasingly complex, more so than necessary, which inevitably increases the risk of exploitable vulnerabilities into the system [7].

The direction Babel goes is exactly the opposite. As we mentioned in Section 1, our goal is to reduce the complexity of the software on the client side, ideally by reducing the trusted code base on the client side.

Additionally, having both client and server in the same machine, isolated by different system VMs, requires a great deal of computer power on the user machine. Some complex code transformation might not be possible if we consider a more limited device. In contrast, having code transformations happening in the cloud gives the opportunity to augment the user's device computing resources, such as CloneCloud [28]. In Babel, however, the augmentation might be limited by the amount of data that is leaked to the security provider, whereas CloneCloud requires that a copy of the device's data exists in the cloud.

[3]Smith and Nair actually use the term "process VM" instead of "application VM" in [27]. Those terms are actually considered synonymous, but we personally prefer and use the latter.

[4]An historical account of the beginning of VMWare in 1999, including their technical challenges, can be found in [8].



```
mov eax, 0xAA
push eax
mov ebx, 0x42
xor eax, eax
add eax, ebx
pop ebx
```

```
mov eax, 0x42
mov ebx, 0xAA
```

Figure 2: Example of optimization by code processing on the server side.

2.3 Code interpretation

Code interpretation in Babel happens within the application VMs. But, instead of a simple and direct interpretation, the code is first sent to the security provider for translation. Only after the translation is received, the VM can interpret the code.

Essentially, in Babel we have a form of JIT compilation that is performed off-site. The original code in the client computer (despite what format it is represented in) can be loosely considered *"the source code"*, that is compiled/translated on demand by the server and subsequently executed by the client.

Of course, one could argue that the translation could occur locally instead of remotely, in order to improve performance. However, this would defeat the whole purpose of translation, which is ensuring that each instruction, or the code execution in general, is analysed by a third party before it can be executed.

Instead, we envision that some (minor) processing might occur on the server side. That would occur, of course, under rather limited conditions. First, either data from the user might not be necessary for the computation or the user must agree that that portion of the data can be sent to the server. An example of the former is a series of operations on constants or on direct memory addresses, as in our (rather contrived) example in Figure 2. Notice that such optimizations, although they might be limited to particular situations, could avoid unnecessary communication. These optimizations might also result in generating a normalized version of the code, for example by removing sequences of redundant instructions or even sequences of *NOOP*, which could even assist the detection mechanism to identify obfuscated malicious code. And the second condition, suggested in [9], is that the time spent performing the computation on the server side and then sending the optimized version must compensate the time of directly sending the non-optimized version of the code to the client.

Also, we must emphasize that our "remote compiler" has also the task of dynamically performing a series of security checks on the code. Those checks might even result in other transformations happening to the code. For example, a sequence of instructions can be dynamically altered to include protection against buffer overflow on an otherwise vulnerable code.

3 Other remote detection

In this section, we initially describe some work that has some similarities to Babel or that are otherwise related in some sense. We then compare them to Babel.

CloudAV takes anti-virus to the cloud [21, 22]. Their approach essentially consists of sending files to a remote server, where the file is checked using a set of different detection engines[5], i.e., different anti-malware products. There are two main insights from CloudAV that inspired our work: the first is that by moving the detection to the cloud, any infection on the client machine will not affect the detection; and the second is that CloudAV uses multiple mechanisms for detection, thus likely covering a wider spectrum of potential threats.

While CloudAV is an inspiration for Babel, the goals and approaches are quite different. As Babel, CloudAV also is composed of a client and a network service component. CloudAV's client component is described as a "lightweight cross-platform host agent" that selects and sends files, usually executables and documents, to the network service. Babel's client, on the other hand, is the actual operating system on the user's machine, or a module in that operating system's kernel, thus Babel is much more integrated into the system. CloudAV's network service is responsible for running a series of detection engines and checking whether the file is infected with some malware, whereas Babel's network service dynamically analyses the actual code executing on the user's computer.

One of the drawbacks of the CloudAV approach is that only files are sent to be tested by the service. There is no assurance that the file was not executed yet, so CloudAV might not avoid a prior infection. CloudAV does not deal with dynamic inputs either. Some of the detection engines include a sandbox for executing the suspicious program, in which case a dynamic detection is performed, but it is still somewhat independent of what can actually happen on the user's side. An example is a long running network service (e.g., an HTTP server) on the user's machine. CloudAV would not be able to detect or deal with a malicious input.

By contrast, Babel aims to detect and avoid malicious activities during code execution. Even though the Babel security provider, which runs the servers, might not have direct access to the actual input, the sequence of instructions or even a sequence of system calls might reveal a malicious activity taking place [15]. Nevertheless, we have also considered some alternative approaches, such as requesting a hash of the input, and check that hash against a database of malicious inputs, or performing some checks on the user's machine, by embedding tests in the code to be executed.

Paranoid Android [25] is a solution especially focused on smartphones. The Paranoid Android (PA) architecture consists of a *tracer* on the user side, i.e., on the user's smartphone, and a *replayer* located in the cloud. The tracer collects information about operations that are performed on the user's smartphone and

[5] In the experiments described in [21], the authors used a total of 12 detection engines.

sends that information to the replayer. The replayer, as the name suggests, replays the sequence of operations while it performs security checks to identify whether that sequence is malicious.

To perform properly, the replayer needs access to the user's data, all of it. PA, therefore, requires that an entire copy of the user's data is sent to the cloud, or, specifically, to the server that runs the detection mechanism. While this need for copy is featured as "transparent data backup", therein lies an important difference from Babel: in our approach, the user is the owner of the information, and it is under the user's discretion how much information is going to be leaked to the cloud.

Another important difference is that PA consists essentially of a post-mortem detection. Whenever a malicious activity is detected by the system, it has happened already. At that point, the authors suggest that the user be warned and that a recovery procedure be started. Such recovery process, however, is not trivial and not really discussed by the authors.

Both CloudAV and PA, and also Babel, are based on a client-server architecture. An interesting point is that CloudAV and PA architectures can be implemented using the Babel infrastructure, with the advantage that we could add earlier detection to PA. The obvious disadvantage of implementing such systems over Babel is that we would lose the control of the user's data. As in the original PA implementation, we would need to send all of the user's data to the cloud.

4 Babel and remote computing

Remote computing refers to the ability of using a remote computer to perform data processing on behalf of the user. Remote computing has taken different forms during the evolution of computers: from *dumb terminals*, which were simply an interface to a remote central computer, to advanced document processing (such as text, spreadsheets) *in the cloud*. Two forms of remote computing are particularly important when discussing Babel.

The first form of remote computing we will discuss is called *thin client computing (TCC)*. Some authors (e.g., [4,18]) have described TCC as a client-server architecture where a *remote display protocol* is used for communication. The main characteristic of a remote display protocol is that it only transmits information about the interface, all the application logic resides on the server.

The second form of remote computing is called *Software-as-a-Service (SaaS)*, which is a type of cloud computing. Cloud computing, by itself, is a rather foggy concept, with each author giving a different definition of what cloud computing means, as captured by Weiss' description of cloud computing:

> *"(...) like the clouds themselves, 'cloud computing' can take on different shapes depending on the viewer, and often seems a little fuzzy at the edges."* [31]

SaaS consists of providing on demand access to software over the cloud. Some of the examples of SaaS are Google Maps, Google Apps, and Apple iCloud. Other forms of cloud computing include PaaS, IaaS and a number of other "X-as-a-Service" [2]. However, for the purpose of this paper, we will limit the discussion to SaaS.

Both SaaS and TCC have the same common characteristic: program execution, for its most part, does not occur on the user's computer, but on a remote site. That is the basic definition of remote computing. In this common characteristic we can already find a fundamental difference from Babel. In Babel, the application logic remains on the client and the actual program execution happens on the client side. While we envision that some portions of the code will be processed in the cloud in order to improve performance, the application logic will still belong to the client. It is the client side that will control how the code will be executed.

Using the traditional comparison between a program and a cooking recipe, we can make the following analogy. With remote computing, we ask a cook to prepare a meal for us. We might know what the ingredients are and have an idea of how the meal is prepared, but we do not know the actual recipe and the cook uses the ingredients that he or she has. In Babel, we have the recipe, but we cannot understand it, maybe because it is in a foreign and strange language. So, we ask an interpreter for some help translating. Because this interpreter happens to know something about cooking, the interpreter also checks if we are doing it right. However, in Babel, we use our own ingredients and do everything ourselves.

There are more subtle, but not less important, differences between TCC, SaaS and Babel. These differences have to do with location and ownership of data and programs. Location indicates where data is stored and handled and whether programs are stored and executed on the "client" or on the "server" side of the model. We classified ownership as "user" or "provider" to indicate who owns the data or programs. We also used "shared" to indicate a situation where the limits for location or ownership are somewhat unclear.

Starting from a very basic example: In a local environment, without network access, it is clear that data and programs belong to the user and both data and program are located on the client side, i.e., the user's computer.

In the case of TCC, what we generally see is that the provider (usually a company) allows the user to access the provider's computer, but usually both data and programs belong to the provider and are located remotely. In fact, one of the appeals for a company to adopt thin clients is to keep its data safe when the data needs to be accessed off-site [17].

With SaaS, the data usually belongs to the user (e.g., documents, photos), but it is also usually located remotely. Programs are obviously located remotely, but their ownership is not really well defined. Programs either belong to the provider or they are "leased" in some form to the user. In fact, Armbrust et al. [1] adds, to the definition of SaaS, programs that run locally but with licensing being enforced remotely.

Babel stands apart from these two. In the Babel model, data and programs

Table 1: Differentiating Babel from remote computing.

Type	Data		Program	
	Ownership	Location	Ownership	Location
Local	User	Client	User	Client
TCC	Provider	Server	Provider	Server
SaaS	User	Server	Shared	Shared
Babel	User	Shared	User	Shared

belong to the user. They are installed and stored locally (unless the user explicitly wants to store data remotely) just as in a local environment. However, as the server's role is to assist the local execution, some parts of the code need to be sent to the server. Portions of data might also be eventually sent, either when necessary in the detection process or for improving the performance of the system, as discussed in Section 2.3. Because data and programs might be shared, it is safer to consider that both data and programs have their location divided between the client and the server side of the architecture.

Table 1 summarizes how location and, more importantly, ownership can be used to differentiate Babel from remote computing. It is clear that, despite being based on a network service, Babel's architecture is closer to a local environment when we consider the ownership of programs and of the user's data.

5 The new world of Babel

Babel presents an increased protection against execution of malicious code. This protection comes in two forms. First, because of the embedded program diversity, an attacker is required to "guess" or somehow identify which language is used by that particular instance of program that is under attack. Second, even if the attacker succeeds in identifying the language and injecting meaningful code into the computer's memory, the code also needs to be checked and attested by the security provider, prior to its execution.

Another important aspect of Babel is its focus on having a very small trusted code base. Our ideal deployment consists of a microkernel architecture. This means that only a few components need to be actually trusted, compared to a monolithic kernel. This small number of components would result in a significantly smaller code base that would need to be audited, thus making feasible a complete audit of the code. The ultimate consequence is that we could, essentially, have an operating system kernel that, after auditing, would not need any updates. And with Babel's ability to perform code transformations, we could have security fixes for any other components of the operating system (and other programs) being implemented *on-the-fly*, protecting the user even more.

However, the enemy is usually persistent. Babel's protection against attacks to the client would actually force attackers to shift their attention from the users to the communication channel and/or to the security provider's servers.

The need for communication becomes an obvious single point of failure in Babel, which makes the communication channel the most obvious target for an attacker. An attacker might want either to disrupt the communication (causing a denial-of-service), or be perceived as the security provider: spoofing and man-in-the-middle attacks would likely be some of the options. In fact, mitigating attacks to the communication channel is also an open problem as new forms of attacks emerge constantly.

We believe that, for the moment, the conventional mechanisms of encryption and authentication are still the best candidates to thwart attacks to the communication channel in Babel as well. The caveats are the extra overhead to an already heavy load, and the need for an infrastructure of trust in order to support authentication. But trust is already an intrinsic part of Babel and is required for the establishment of security providers, thus the authentication might actually come more naturally than in the more general case of the internet.

A secondary target would the security providers themselves. However, that means a very high profile target. It would not only require a very talented attacker (or team of attackers), but also much more resources than is necessary nowadays to attack generic internet users. The game will become harder for the attackers. And it is the security provider's responsibility to keep a well maintained infrastructure to support its users.

It is clear that Babel's co-dependency requires a great deal of communication, especially for translating program instructions. This communication, and more particularly the latency imposed by the network environment, might make execution too slow and unbearable.

Therefore, a full adoption of Babel presupposes a world where latency does not exist or it is minimal. While in the last 20 years we have seen over 1000% improvement in terms of network bandwidth [23], latency is still a problem to be solved [26].

But we cannot simply wait for a solution. So, in order to make Babel usable in the near future, even while that extremely low latency is still not available, we have been working on several approaches to mitigate the problem. We have already reduced significantly the number of round-trips by implementing a small cache of translated instructions and by sending blocks of instructions for translation [3]. Further work will also explore branch prediction [19] so we can anticipate instructions that might need translation shortly.

Also, while Babel is definitely not SaaS, we want to leverage the benefits from cloud computing. Cloud computing can contribute to reduce latency in at least two more ways [30, p. 266-7]. First, by leveraging its geographically distributed aspect, which increases the chances of bringing the service closer to the user, no matter where in the world the user is. Second, by using its inherent parallelism, which can potentially improves response time by dividing complex tasks such as code optimization and the security verification.

Another natural point to be considered is user's privacy. While we attempt to avoid or minimize data leakage, the fact that all the computation is overseen by an external entity certainly raises concern. In [3], we discuss some possible ways to mitigate the loss of privacy. Nevertheless, we admit that a more thorough investigation on how Babel impacts privacy and on how to deal with that impact is still necessary. However, we believe that the existence of a fully developed Babel system would be required, or at least desired, to provide grounds for such investigation.

Babel, thus, presents a different and interesting approach for computer security; an approach with great potential, in which still unknown defense mechanisms (and, of course, new attack methods) wait to be unveiled.

6 Acknowledgments

The author's research is supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada via ISSNet, the Internetworked Systems Security Network. Thanks to my supervisor John Aycock, and to Michael Locasto and Chris Jarabek, also co-authors of [3], for helpful discussions, ideas and suggestions. Also, I would like to thank the all the attendees of NSPW 2013, for their thoughtful suggestions and feedback.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, February 2009.

[3] J. Aycock, D. M. N. Nunes de Castro, M. E. Locasto, and C. Jarabek. Babel: a secure computer is a polyglot. In *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop*, CCSW '12, pages 43–54, New York, NY, USA, 2012. ACM.

[4] R. A. Baratto, L. N. Kim, and J. Nieh. THINC: a virtual display architecture for thin-client computing. In *Proceedings of the 20th ACM symposium on Operating systems principles*, SOSP '05, pages 277–290, New York, NY, USA, 2005. ACM.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[6] S. W. Boyd, G. S. Kc, M. E. Locasto, A. D. Keromytis, and V. Prevelakis. On the general applicability of instruction-set randomization. *IEEE Transactions on Dependable and Secure Computing*, 7(3):255–270, July 2010.

[7] S. Bratus, M. E. Locasto, A. Ramaswamy, and S. W. Smith. VM-based security overkill: a lament for applied systems security research. In *Proceedings of the 2010 New Security Paradigms Workshop*, NSPW '10, pages 51–60, New York, NY, USA, 2010. ACM.

[8] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Transactions on Computer Systems*, 30(4):12:1–12:51, 2012.

[9] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: elastic execution between mobile device and cloud. In *Proceedings of the 6th ACM European conference on Computer systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[10] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02, pages 211–224, New York, NY, USA, 2002. ACM.

[11] M. W. Eichin and J. A. Rochlis. With microscope and tweezers: an analysis of the internet virus of november 1988. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pages 326–343, May 1989.

[12] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[13] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, HOTOS '97, pages 67–, Washington, DC, USA, 1997. IEEE Computer Society.

[14] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS '03. The Internet Society, 2003.

[15] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, August 1998.

[16] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and Communications Security*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM.

[17] J. Krikke. Thin clients get second chance in emerging markets. *Pervasive Computing*, 3(4):6 – 10, October–December 2004.

[18] A. M. Lai and J. Nieh. On the performance of wide-area thin-client computing. *ACM Transactions on Computer Systems*, 24(2):175–209, May 2006.

[19] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22, 1984.

[20] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[21] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-version antivirus in the network cloud. In *Proceedings of the 17th USENIX Security Symposium*, pages 91–106, Berkeley, CA, USA, 2008. USENIX Association.

[22] Jn Oberheide, E. Cooke, and F. Jahanian. Rethinking antivirus: executable analysis in the network cloud. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Security*, HOTSEC '07, pages 5:1–5:5, Berkeley, CA, USA, 2007. USENIX Association.

[23] D. A. Patterson. Latency lags bandwith. *Communnications of ACM*, 47(10):71–75, October 2004.

[24] R. Pike. The design of the Inferno virtual machine. `http://doc.cat-v.org/inferno/4th_edition/dis_VM_design`. Last accessed: 07.Sep.2012 .

[25] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 347–356, New York, NY, USA, 2010. ACM.

[26] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's time for low latency. In *Proceedings of the 13th USENIX conference on Hot Topics in Operating Systems*, HotOS '11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[27] J. E. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005.

[28] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[29] S. Wegscheider-Cruse, J.R. Cruse, and J. Cruse. *Understanding Co-Dependency*. Health Communications, Inc., 1990.

[30] J. Weinman. *Cloudonomics: The Business Value of Cloud Computing*. Wiley Publishing, 1st edition, 2012.

[31] A. Weiss. Computing in the clouds. *netWorker*, 11(4):16–25, 2007.