Speculative Virtual Verification: Policy-Constrained Speculative Execution

Michael E. Locasto
Network Security Lab
Department of Computer Science
Columbia University
locasto@cs.columbia.edu

Stelios Sidiroglou
Network Security Lab
Department of Computer Science
Columbia University
ss1759@cs.columbia.edu

Angelos D. Keromytis
Network Security Lab
Department of Computer Science
Columbia University
angelos@cs.columbia.edu

ABSTRACT

A key problem facing current computing systems is the inability to autonomously manage security vulnerabilities as well as more mundane errors. Since the design of computer architectures is usually performance-driven, hardware often lacks primitives for tasks in which raw speed is not the primary goal. There is little architectural support for monitoring execution at the instruction level, and no mechanisms for assisting an automated response.

This paper advocates modifying general-purpose processors to provide both program supervision and automatic response via a policy-driven monitoring mechanism and instruction stream rewriting, respectively. These capabilities form the basis of *speculative virtual verification* (SVV).

SVV is a model for the speculative execution of code based on high-level security and safety constraints. We introduce architectural enhancements to support this framework, including the ability to supply an automated response by rewriting the instruction stream. Finally, given the novelty of the SVV approach to executing software, we briefly consider some important challenges for SVV-based systems.

Categories and Subject Descriptors

B.8 [**Hardware**]: Performance and Reliability
; C.1.3 [**Processor Architectures**]: Other Architecture Styles

General Terms

Security, Design, Reliability

Keywords

SVV, Speculative Execution, Micro-speculation, hardware security

NSPW 2005 Lake Arrowhead CA USA

1. INTRODUCTION

Software faults and vulnerabilities continue to present significant obstacles to achieving reliable and secure software. The lack of comprehensive and low-cost protection mechanisms presents a critical problem for computing systems.

Static analysis techniques or improved programming practices are unlikely to provide a complete solution to the types of errors that threaten system stability or create exploitable vulnerabilities. Even systems that dynamically monitor process execution often impose a noticeable performance cost. Furthermore, these systems may reinvent the same primitives because the hardware does not supply them. However, even if such capabilities existed, system security is often a matter of *policy*; these utilities would need some level of flexibility to be applicable and remain useful in a wide variety of diverse and evolving environments. Finally, systems currently lack the capability to respond intelligently to both attacks and non-malicious faults.

The ability for computing systems to detect and correct faults and vulnerabilities would greatly improve their stability and security. The main contribution of this paper is the proposal of a set of architectural components that provide a basis for such systems by speculatively executing the entire instruction stream. In much the same way that a superscalar processor speculatively executes past a branch instruction and discards the mis-predicted code path, we propose that processors operate on the instruction stream in two phases. The first phase executes instructions, optimistically "speculating" that the results of these computations are benign. The second phase makes the effects of the speculated instruction stream visible to the OS and application software layers and potentially rewrites the instruction stream if it has been deemed harmful.

1.1 Speculative Execution

Speculative execution is a technique used in microprocessors to execute the instructions in a code branch before the evaluation of the branch conditional is finished. The need to perform speculative execution arises in pipelined processors because the conditional instruction that the branch depends on has proceeded deeply into the pipeline but has not been evaluated by the time the processor is ready to fetch additional instructions. An example of this situation is shown in Figure 1.

While a complete discussion of the strategies for dealing with branch predication is beyond the scope of this paper,

```
0  ...
1    fdivl  %R1, %R2, %R3
2    fadd   %R5, %R6, %R7
3    fcmp   %R1, %R2
4    je     LABEL1
5    jmp    LABEL2
6  LABEL1:
7    movl   $0, %R1
8    movl   $0, %R2
9    movl   %R1, -8(%R30)
10   movl   %R2, -4(%R30)
11   jmp    LABEL3
12 LABEL2:
13 ...
```

Figure 1: *Speculative Execution of a Branch.* In this made-up assembly language, rather than stall the pipeline because of the unresolved result of the floating point divide operation, the processor can choose to issue the floating point add operation on line 2 (out of order execution). If the dependency on R1 and R2 between the divide and the compare operations is satisfied, then the compare can execute. Because the result of the compare on line 3 may not be available for the branch instruction in line 4, the processor may speculatively execute (based on branch prediction) the code at LABEL1 or fall through to the direct jump on line 5. If the branch prediction is incorrect, the speculated instructions are flushed and execution continues from the correct target.

a basic overview of the subject and pointers to other material are available in [11, 7]. Our proposal differs from these techniques by introducing an additional layer of speculative execution in which the acceptance of a particular execution path is not based on the evaluation of a branch conditional, but rather a higher-order constraint on a set of instructions.

1.2 Motivation and Feasibility

We are motivated by work on constructing an emulator [29] to supervise program execution in response to exploits and errors. Unfortunately, the use of an emulator imposes a considerable performance overhead since the emulator executes every program instruction in software. The first way to ease this burden, which was adopted in [29], is to limit the scope of emulation to portions of the program demonstrated to be vulnerable, thereby reducing the time that is spent in the emulator. The second approach is to eliminate the emulation penalty altogether by executing the process directly on the CPU. Unfortunately, adopting this approach currently means relinquishing the monitoring capabilities that the emulator provides. Therefore, we advocate adding monitoring mechanisms to processors so that a certain level of safety is relatively inexpensive. In order to address more complex attacks, we also propose that execution can be delegated to the software emulator as needed.

Our goal is to push common-mode security monitoring functionality further down the system stack. Arguing for the widespread adoption of fundamental changes to hardware is a controversial proposition. We believe the hardware necessary to support our system is easily implementable. Indeed, large parts of the system are already present in modern processors to support thread level speculation (TLS). The design parameters of general-purpose microprocessors have traditionally been driven by raw performance. We advocate design parameters aimed at more high-level feature support.

2. SYSTEM DESIGN

As illustrated in Figure 2 the core features of SVV form a two level monitoring environment. The first level includes hardware mechanisms for monitoring instruction execution (bounds checking, taint-tracking [32], SRAS [16], transfer control validation [14], *etc.*). The second level of monitoring is provided by the Policy Constraint Unit (PCU) and the Virtual Emulator Registration Unit (VERU). Instructions are filtered by the PCU according to some policy constructed by the programmer, compiler, or runtime profiling. The policy could range from filtering on a particular class of instructions (integer vs. load/store) to more complex constraints that require keeping state. The design of a constraint language to express these policies is future work, but we envision the PCU to be a FSM much like the instruction decoding unit that is able to filter instructions based on properties like target and source registers and memory locations, instruction type, and processor status flags, other processor state (as supplied by other components such as a SRAS or an array length tracker), and data dependencies.

The VERU stores an address for code that should be executed if the PCU identifies a sequence of instructions that require more resources than the hardware can easily provide. Finally, the Verification Buffer (VB) and the Instruction Rewrite Unit (IRWU) provide some basic support for an automatic response capability.

2.1 SVV Execution Model

The execution model for SVV (see Figure 2) is similar to current superscalar execution models. Instructions are fetched, decoded, issued to functional units (possibly out of order), executed, and gathered in a re-order buffer (ROB) to be committed in program order. However, at each stage, instructions are filtered by the PCU and monitored by hardware-level security mechanisms. Additionally, the VB accumulates completed instructions as they leave the ROB and commits them only if they pass the monitoring tests.

Instruction flow for SVV can be categorized by the following three scenarios. First, the instruction may be harmless. In this case, it proceeds normally to the ROB, graduates when appropriate, moves to the VB, and is committed. Second, an instruction may be harmful as determined by the monitoring mechanisms (*e.g.*, it is actually tainted input data, or will write input data to the code area of the process address space) or the PCU. In this case, the IRWU flushes the scope of the harmful instruction and constructs a 'safe' version of the flushed code. The processor then executes this alternate instruction stream, including a return to the normal path of execution. The third scenario enables an emulator to be loaded on the CPU and supervise code execution. If the PCU decides that a particular sequence of instructions requires more complex supervision, it can invoke execution of this emulator. Note that there is no requirement for the software invoked by the VERU to be an emulator. The VERU simply holds an address and transfers control to the code at this address. Such an approach enables a more general response mechanism than software emulation. For example, the VERU may transfer control to

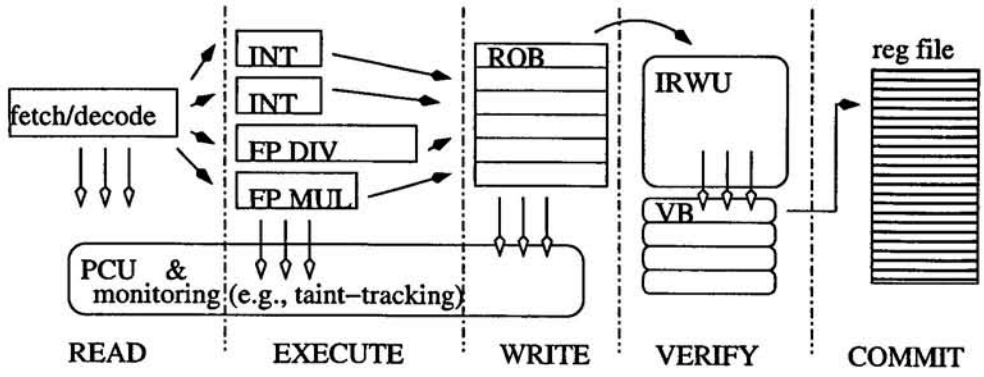

Figure 2: *Pipeline organization for SVV.* Here, a simplified pipeline for a superscalar processor is modified to add an extra verification stage as well as policy-driven hardware-based monitoring mechanisms. The IRWU can optionally rewrite the instruction stream and cause the new version (stored in the VB) to be executed. Traditional hardware components are shown as full rectangles, new components are rounded. Not shown is the VERU, which holds the address for an emulator capable of higher-level supervision.

an OS routine that kills the process, or suspends the process and transfers it to an isolated host for analysis, auditing, intrusion detection, or debugging.

Another way to envision the SVV execution model is as an operating system that schedules a process for execution on two cooperating microprocessors, as shown in Figure 3. Such an implementation would needlessly complicate the OS, and we argue that individual processors can contain the components necessary to transparently implement SVV.

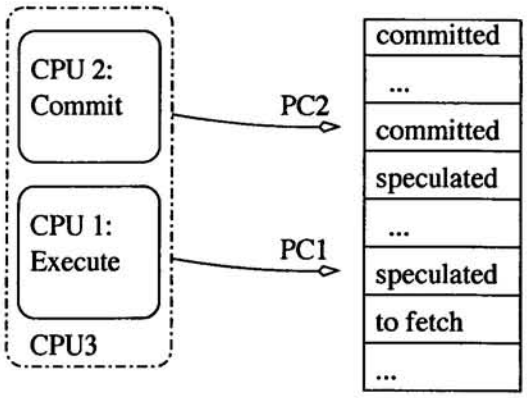

Figure 3: *High-level execution model for SVV.* The instruction stream for a process is scheduled for execution on two processors. CPU1 supervises instruction execution while CPU2 commits instructions that are benign. CPU2 can optionally re-write the instruction stream as a basic form of active response. The conceptual processors CPU1 and CPU2 are actually one physical unit, CPU3.

2.2 Scope of SVV

The largest obstacle to overcome for SVV is a three part problem that involves determining the scope of supervision. First, even though SVV is meant to run continuously, some applications (especially those working in a power-constrained environment) may wish to avoid the overhead associated with constant monitoring. Second, hardware is fundamentally limited in the number of virtual execution contexts it can support concurrently. Finally, it is likely that the basic monitoring mechanisms, while capable of stopping large classes of attacks, may be unable to cope with more sophisticated attacks (some forms of DoS, multi-step attacks, information leaks, improperly set permissions, phishing attacks, *etc.*) or analysis tasks that require copious amount of state (anomaly or intrusion detection via data mining).

To address the latter two problems, we use the VERU to register a software emulator that can perform high-level monitoring of an instruction stream. An emulator has the flexibility to be more intrusive and is easily customizable. This hybrid approach to monitoring is more promising than an approach based solely on hardware or software. To address the first problem, SVV can be selectively invoked. Control over this invocation can be handled by the OS (a new system call to invoke or halt the SVV hardware) or the compiler (new assembly instructions can delimit an SVV monitored code region).

2.3 Micro-patching: Automated Response

Automating a response strategy is difficult, as it is often unclear what a program should do in response to an error or attack. A response system is forced to anticipate the intent of the programmer, even if that intent was not well expressed or even well-formed. Ideal computing systems would recover from attacks and errors without human intervention. However, the state of the art is far from mature, and most existing response mechanisms are external to the system they protect. Some simply crash the process that was attacked (and do nothing to fix the fault, thereby ensuring that the system is still vulnerable when it is rebooted). Other systems may restrict network connectivity or resource consumption. SVV includes the ability to rewrite a vulnerable sequence of instructions without recompilation. In effect, SVV supports the ability to generate and insert a micro-patch into the protected application's instruction stream.

This mechanism is general enough that a wide variety of response techniques can be implemented, such as: data

structure repair [6], failure oblivious computing [26], and error virtualization [29]. Compilers can be augmented to provide "alternative execution paths" to some code sections. These alternatives can be driven by explicit program code, programmer annotation, purely compiler-generated, taken from profiling information for the application, or gathered by the processor itself from previous runs of the same code block as a form of machine learning.

The rewritten instruction stream can be propagated to the code section of the process address space to protect future execution. The new instruction sequence could be applied (with OS support) to the on-disk binary as a rudimentary patch. The question of whether or not to propagate the micro-patch out to the process memory space or even to disk is a high-level policy question. One difficulty with automatically propagating the patch (beyond the current invocation) is that attacks and faults are relatively rare, and executing the micro-patch for all subsequent normal requests would needlessly change the normal operation of the software. One solution is to have a prologue to all micro-patches such that they are conditionally executed based on site policy (as set by an administrator who knows the needs of the environment). Another solution is to have the micro-patch conditionally executed based on markers seen in the environment. For example, at the moment of patching, a software-level monitor can take a snapshot of important state (network packets seen, important data structures), and if those conditions are recreated, the monitor can set a flag so that the micro-patch does execute.

Micro-patching via instruction stream rewriting can be seen as a type of automatic diversity mechanism. While automated diversity is a good protection mechanism, we argue that micro-patches should be recorded somewhere (even if they are not automatically propagated to the process image or binary); failing to do so can make it difficult to debug an application, as there would be no exact record of what code the processor generated and executed.

There are many pitfalls to automating a response. One interesting possibility is for an attacker to implement a covert channel by continuously causing SVV to flush the current set of instructions and replace it with a micro-patch. Such an attack would seem to be difficult, as the current execution context (and thus, presumably the attacker's code) would be replaced with completely different instructions, but it not at all outside the realm of possibility. The micro-patch itself would have to cause an externally measurable phenomena for the consumer of the covert channel.

3. RELATED WORK

SVV draws on ideas from computer architecture, fault-tolerant computing, and computer security. We examined some hardware support [30] for an *x86* emulator (STEM) that supervises the execution of vulnerable code slices [29]. The approach of SVV is akin to systems [28, 25, 23] that utilize a secondary host machine as a sandbox or instrumented honeypot: work is offloaded to this host, thus minimizing exposure to the primary host. The work most closely related to ours is Oplinger and Lam's proposal [22] for using TLS to improve software reliability. Their key idea is to execute an application's monitoring code in parallel with the primary computation and roll back the computation "transaction" depending on the results of the monitoring code.

Evers *et al.* [7] investigate the predictability of branches and provide an overview of various branch prediction schemes that have been proposed to ameliorate the cost of incorrect predictions. Wang *et al.* [33] explore an interesting result: about 50% of mispredicted branches do not affect correct program behavior. This result is encouraging because it offers evidence that our previously proposed macro-level remediation technique of *error virtualization* (dynamically returning early from the current function context with an extrapolated error code) holds at the micro-level also.

3.1 Secure Hardware

Incorporating security mechanisms in hardware has traditionally been limited to providing implementations of cryptographic algorithms. McGregor and Lee [20] also investigate protecting cryptographic secrets. Of a more focused scope is Lee *et al.*'s proposal [16] of a hardware-based return stack (SRAS) to frustrate buffer overflow attacks. Suh *et al.* [32] propose hardware extensions to thwart control-transfer attacks by tracking "tainted" input data (as identified by the OS). If the processor detects the use of this tainted data as a jump address or an executed instruction, it raises an exception. Kuperman et al. [15] has a good overview of buffer-overflow related attacks and discusses some hardware-based approaches to protection, including SRAS (and related variants) and their own SmashGuard proposal.

Even contemporary approaches to this topic, such as the TCPA/TCG, only provide tamper-resistant hardware modules to store secrets. Recent efforts such as Cerium [4] and XOM [18, 17, 19] focused on providing a trusted computing base (TCB) and tamper-resistant architecture that can attest to the validity of a particular computation [10]. In the case of execute-only memory (XOM), the hardware performs encrypted program execution and makes several strong security claims.

While TCG does offer some measurement functionality [27], the state of the art in this field tries to leverage these stored secrets for attestation, and attestation is typically used for the purposes of DRM. Such uses provide a mechanism for a remote entity to control local execution. There are no mechanisms for the local entity to systematically prevent and control a remote entity from executing local code. Our work on SVV is an attempt to provide a unified model for the supervision and online patching of machine instructions.

The Copilot system [24] by Petroni et al. is one expression of hardware security aimed at integrity protection. Much like the Tripwire[1] software, the goal of Copilot is to make sure that important data has not been corrupted. However, Copilot performs rootkit intrusion detection by monitoring changes to a host's kernel text segment and related data structures. The current implementation is based on a PCI card that monitors the host's main memory via DMA (without the host kernel's knowledge) and has a secure communications link to an administrative reporting station.

3.2 Execution Supervision Environments

Virtual machine emulation of operating systems or processor architectures to provide a sandboxed environment is an active area of research. Virtual machine monitors (VMMs) are employed in a number of security-related contexts, from autonomic patching of vulnerabilities [28] to intrusion detection [9]. MiSFIT [31] is a tool that constructs a sand-

[1] `http://tripwire.org/`

box by instrumenting applications at the assembly language level. Program shepherding [14] works on uninstrumented IA-32 binaries and validates branch instructions to prevent transfer of control to injected code. Intel's Vanderpool and AMD's Pacifica designs are forward-looking architectures that provide support for hypervisors and VMMs. These designs provide the mechanisms we wish to use for the support and invocation of the Virtual Emulator for more high-level monitoring.

Other protection mechanisms include compiler techniques like Stackguard [5] and safer libraries, such as *libsafe* and *libverify* [1]. Tools exist to verify and supervise code during development or debugging; of these tools, Purify[2] and Valgrind [21] are popular choices. Valgrind has been used by Barrantes *et al.* [2] to implement instruction set randomization (ISR) to protect against code insertion attacks. Other work on ISR includes [13], which employs the *x86* emulator Bochs[3]. The implementation of ISR techniques in hardware would eliminate most of their performance impact.

In work inspired by the ideas fundamental to artificial system diversity [8], Holland, Lim, and Seltzer [12] introduce the idea of automatically generating randomized architectures to support system security. Since synthesizing the hardware for such every such generated architecture is an untenable approach, they recommend using VMMs to provide the necessary execution environments.

3.3 Recovery and Repair

A key feature of SVV is the use of instruction stream rewriting as a basic building block for an adaptive response mechanism. Other recent work that examines repair mechanisms includes failure-oblivious computing [26] and data structure repair [6]. Candea and Fox propose a different approach: design software systems such that they employ crashing as the normal halting mode and use recursive microreboots to safely restart [3]. We propose adding the capability to rewrite local code slices in the processor itself as a general tool for reactive capabilities.

4. CONCLUSIONS

We have described the architectural components needed to support a new execution model for secure and reliable computing: *speculative virtual verification* (SVV). This model complements previous work on trustworthy and tamper–resistant computing architectures but is not meant as a replacement for the capabilities such systems provide. There is a multitude of challenging problems to be addressed in the construction, testing, and deployment of SVV. We intend to study these issues and implement SVV in a variety of execution environments, including *x86* emulators, the Java Virtual Machine, and simulators for the MIPS and ARM architectures.

There is no silver bullet for system security, and SVV is not meant to address all possible attacks. However, we believe that given the current state of the arms race between attackers and system designers a paradigm shift is necessary. We advocate modifying general-purpose processors to (*a*) provide implicit supervision functionality, (*b*) export a policy-driven monitoring mechanism, and (*c*) provide the foundation for an automatic response capability via instruction stream rewriting.

[2] `http://www.rational.com/products/purify_unix/index.jtmpl`

[3] `http://bochs.sourceforge.net/`

Acknowledgments

Few papers can claim to be the product of one mind. In this case, the authors would like to thank the anonymous reviewers for providing an example of how high-quality the peer-review process could be. We would also like to thank the attendees of NSPW 2005 for providing thoughtful comments on the interesting aspects of SVV, praise for the novel parts, and gentle criticism for the incomplete parts. Finally, Carrie Gates and Bob Blakley deserve special recognition for serving as scribes for not just this paper, but the whole workshop.

5. REFERENCES

[1] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[2] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Distrupt Binary Code Injection Attacks. In *10th ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[3] G. Candea and A. Fox. Crash-Only Software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HOTOS-IX)*, May 2003.

[4] B. Chen and R. Morris. Certifying Program Execution with Secure Processors. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, pages 133–138, May 2003.

[5] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. 1998.

[6] B. Demsky and M. C. Rinard. Automatic Data Structure Repair for Self-Healing Systems. In *Proceedings of the 1st Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.

[7] M. Evers, S. J. Patel, and Y. N. Patt. An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[8] S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[9] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In 10th *ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, February 2003.

[10] T. Garfinkel, M. Rosenblum, and D. Boneh. Flexible OS Support and Applications for Trusted Computing. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, pages 145–150, May 2003.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 3rd edition, 2003.

[12] D. A. Holland, A. T. Lim, and M. I. Seltzer. An Architecture a Day Keeps The Hacker Away. In *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, October 2004.

[13] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *10th ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[14] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[15] B. A. Kuperman, C. E. Brodley, H. Ozdoganoglu, T. N. Vijaykumar, and A. Jalote. Detection and Prevention of Stack Buffer Overflow Attacks. *Communications of the ACM*, 48(11):51–56, November 2005.

[16] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi. Enlisting Hardware Architecture to Thwart Malicious Code Injection. In *Proceedings of the International Conference on Security in Pervasive Computing (SPC-2003), Lecture Notes in Computer Science, Springer Verlag*, March 2003.

[17] D. Lie, J. Mitchell, C. Thekkath, and M. Horwitz. Specifying and Verifying Hardware for Tamper-Resistant Software. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2003.

[18] D. Lie, C. Thekkath, and M. Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings of the 19^{th} ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[19] D. Lie, C. Thekkath, M. Mitchell, and P. Lincoln. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the 9^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, 2000.

[20] J. P. McGregor and R. B. Lee. Protecting Cryptographic Keys and Computations via Virtual Secure Coprocessing. In *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, October 2004.

[21] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Electronic Notes in Theoretical Computer Science*, volume 89, 2003.

[22] J. Oplinger and M. S. Lam. Enhancing Software Reliability with Speculative Threads. In *Proceedings of the 10^{th} International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS X)*, October 2002.

[23] H. Patil and C. N. Fischer. Efficient Turn-time Monitoring Using Shadow Processing. In *Proceedings of the 2^{nd} International Workshop on Automated and Algorithmic Debugging*, 1995.

[24] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In 13^{th} *USENIX Security Symposium*, pages 179–194.

[25] J. C. Reynolds, J. Just, L. Clough, and R. Maglich. On-Line Intrusion Detection and Attack Prevention Using Diversity, Genrate-and-Test, and Generalization. In *Proceedings of the 36^{th} Hawaii International Conference on System Sciences (HICSS)*, 2003.

[26] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6^{th} Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[27] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In 13^{th} *USENIX Security Symposium*, pages 223–238.

[28] S. Sidiroglou and A. D. Keromytis. A Network Worm Vaccine Architecture. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security*, pages 220–225, June 2003.

[29] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*, pages 149–161, April 2005.

[30] S. Sidiroglou, M. E. Locasto, and A. D. Keromytis. Hardware Support For Self-Healing Software Services. In *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, pages 37–43, October 2004.

[31] C. Small and M. Seltzer. MiSFIT: A Tool for Constructing Safe Extensible C++ Systems. *IEEE Concurrency*, 6(3):33–41, 1998.

[32] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, October 2004.

[33] N. Wang, M. Fertig, and S. J. Patel. Y-Branches: When You Come to a Fork in the Road, Take It. In *Proceedings of the 12^{th} International Conference on Parallel Architectures and Compilation Techniques*, September 2003.