Toward Application-Aware Security and Reliability

Ravishankar K. Iyer, Zbigniew Kalbarczyk, Karthik Pattabiraman, William Healey, Wen-Mei W. Hwu, Peter Klemperer, and Reza Farivar
University of Illinois at Urbana-Champaign

Two trends—the increasing complexity of computer systems and their deployment in mission- and life-critical applications—are driving the need to provide applications with security and reliability support. Compounding the situation, the Internet's ubiquity has made systems much more vulnerable to malicious attacks that can have far-reaching implications on our daily lives. The catastrophic failure of AT&T's telecommunication network in New York City, for example, which affected both airline reservation and American Red Cross blood-supply-tracking systems,[1] and the Code Red worm, which exploited a buffer overflow in Internet Information Service (IIS), indicate a rising problem with computer and system security.

Traditionally, system security has meant access control and cryptography support, but the Internet's phenomenal growth has led to the large-scale adoption of networked computer systems for a diverse cross section of applications with highly varying requirements. In this all-pervasive computing environment, the need for security and reliability has expanded from a few expensive systems to a basic computing necessity. This new paradigm has important consequences:

- Networked systems stretch the boundary of fault models, from an application or node failure to failures that could propagate and affect other components, subsystems, and systems.
- Attackers can exploit vulnerabilities in operating systems and applications with relative ease.
- As computing systems become more ubiquitous, security and reliability techniques must be cheaper and more focused on application characteristics.

Clearly, the traditional one-size-fits-all approach to security and reliability is no longer sufficient or acceptable from the user perspective. A potentially much more cost-effective and precise approach is to customize the mechanisms for detecting security attacks and execution errors using knowledge about the expected or allowed program behavior. Spectacular system failures due to malicious tampering or mishandled accidental errors call for novel, application-specific approaches. In this article, we introduce the concept of application-aware checking as an alternative. By extracting application via recent breakthroughs in compiler analysis and enforcing the characteristics at runtime with the hardware modules embedded in the reliability and security engine (RSE), it's possible to achieve security and reliability with low overheads and false-positive rates.

Why application-aware?

Users want their applications to continue to operate without interruption, despite attacks and failures, but as applications become more complex and sensitive, this task becomes more difficult. Furthermore, because hardware is becoming cheaper, it's much more desirable for the underlying hardware to configure itself to provide the best support on a per-application basis.

Application-aware checking provides knowledge about an application's characteristics to the underlying hardware. This in turn makes the application aware of underlying hardware techniques that it can invoke at critical points in its execution to request reliability and security support when necessary. As a result, application-aware checking makes attack and error checkers smarter, so that they detect only those errors that affect the application.

The principle of application-aware checking applies at all levels of the system hierarchy—operating system, middleware,[2] and application.[3] The idea of customizing hardware checkers based on application needs is especially compatible with recent industry trends such sa utility computing, in which large hardware farms provide a platform for complex, long-running applications to operate in a timely fashion (see http://devresource.hp.com/drc/topics/utility_comp.jsp and www.network.com). Indeed, utility computing grids can optimize themselves to suit the application's performance needs. Analogously, an application-aware checking methodology lets hardware checkers tune themselves

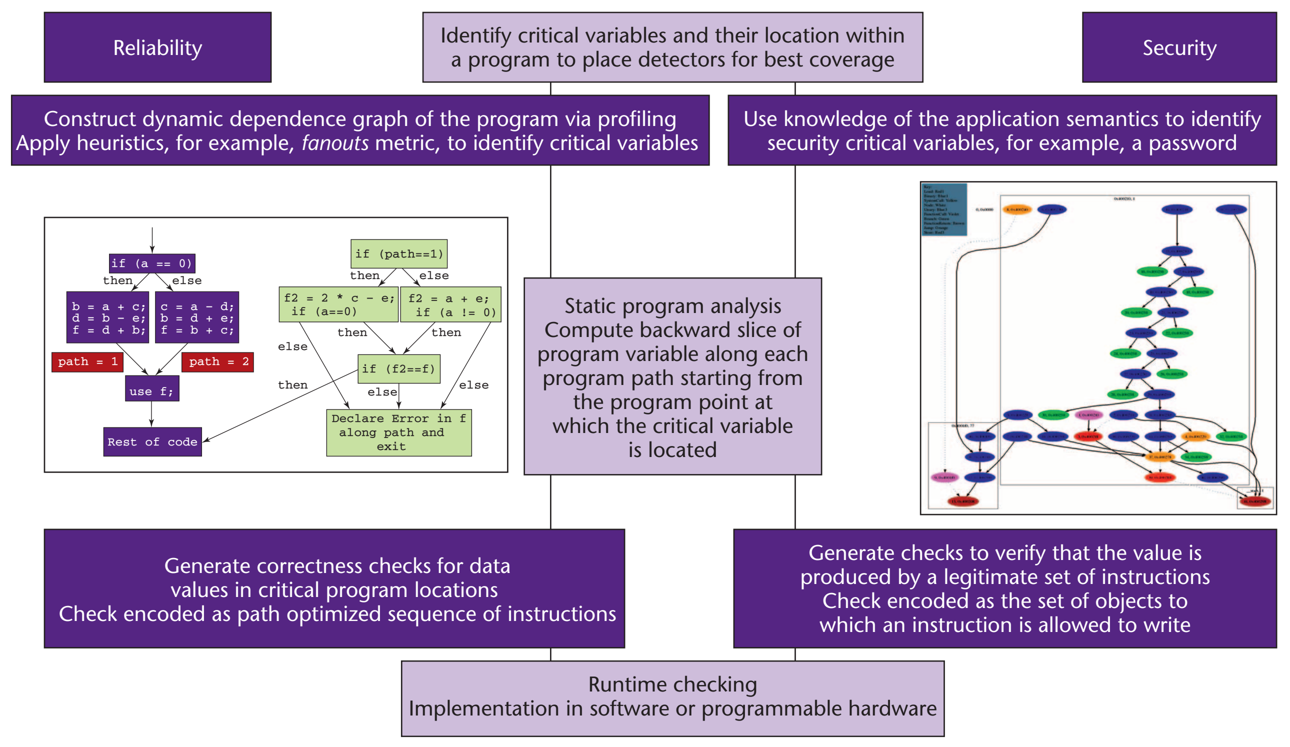
Reliability
Identify critical variables and their location within a program to place detectors for best coverage
Security
Construct dynamic dependence graph of the program via profiling
Apply heuristics, for example, *fanouts* metric, to identify critical variables
Use knowledge of the application semantics to identify security critical variables, for example, a password
if (a == 0)
then
else
b = a + c; d = b − e; f = d + b;
c = a − d; b = d + e; f = b + c;
path = 1
path = 2
use f;
Rest of code
if (path==1)
f2 = 2 * c − e; if (a==0)
f2 = a + e; if (a != 0)
if (f2==f)
Declare Error in f along path and exit
Static program analysis
Compute backward slice of program variable along each program path starting from the program point at which the critical variable is located
Generate correctness checks for data values in critical program locations
Check encoded as path optimized sequence of instructions
Generate checks to verify that the value is produced by a legitimate set of instructions
Check encoded as the set of objects to which an instruction is allowed to write
Runtime checking
Implementation in software or programmable hardware

Figure 1. Common framework for security and reliability checks. After identifying critical variables, the compiler extracts the backward program slice, which serves as a basis for security and reliability checks.

to detect the relevant errors affecting an executing application.

Hardware-based techniques also have the advantage of low performance overhead because the hardware modules can perform security and reliability checking in parallel with the application's execution. With knowledge about allowed application behavior, application-aware hardware techniques can detect errors close to their point of occurrence, and hence low levels of detection latency are possible. This isn't always true for software-implemented application-, middleware-, or operating system-level techniques.

A new framework

Our hardware-based technique uses knowledge of an application's execution characteristics to devise application-specific detectors and assertions for low-latency data-corruption detection. Figure 1 illustrates a framework for automated (or semi-automated) derivation of security and reliability checks.

The framework in Figure 1 uses compiler-based static analysis to uncover relationships or invariants that hold in the original program, so that the hardware can check them during runtime to detect security and reliability violations. An example of an invariant is that only certain instructions can write to a specific memory location; a violation of this invariant signals an error or attack. The first step in static analysis is to identify critical variables and locations in the program, which, if corrupted can lead to failures or security breaches with a high probability. For reliability, the compiler identifies critical variables based on heuristics applied to the program's dynamic dependence graph.[3] For security, programmers use their knowledge of application semantics to identify critical variables—for example, the variable that holds the system password for authentication might be critical in a network communication application.

Security checks

The goal of security checks is to defend against potential security attacks by preventing malicious corruption of critical data (such as passwords). This is accomplished by enforcing the program's semantics upon its execution.

Our broad threat model includes

- *memory attacks*, in which the attacker can execute arbitrary code or overwrite program variables stored in memory and processor registers;
- *physical attacks*, in which code is injected via malicious hardware device; and
- *insider attacks*, in which the attacker tries to alter (at runtime) part of the program to gain control over the application or system.

```
int main()
{
char password[8] = "asecret";
char userpass[8];
printf("Enter Password:\n");
gets(userpass);
if(strncmp(userpass,password,7)==0)
printf("Success\n");
else
printf("Failed\n");
}
```

Figure 2. Example code fragment. The program outputs "Success" or "Failed" when the user provides a correct or invalid password, respectively.

Because we want to ensure the integrity of critical data (rather than ensure data confidentiality), our approach doesn't address side-channel attacks. We also assume that the original program itself isn't malicious, thus we don't consider attacks in which the program's source code is tampered with (such as Trojan horse attacks).

Languages such as C and C++ aren't memory-safe, so any pointer can access (or write) any location in the application's memory space. Consequently, an attacker can exploit memory errors to get access to security-critical data by overflowing nearby memory objects.[4] Buffer overflow attacks, for example, are possible because the system allocates two buffers next to each other on the stack, thus letting an attacker use a pointer from one buffer to access the other. Similarly, in a format-string attack, an attacker can use an ill-formed format-string argument to the `printf` function to write to an arbitrary memory location. The ability to write past the end of an object into other valid objects is an artifact of the runtime system (the linker/loader), not an artifact of source semantics.

Essentially, these attacks exploit memory corruption because of the gap between the program's source- and application-level semantics and its execution semantics and runtime implementation. We've found two security-checking techniques that bridge this gap and enforce either source-level semantics (dataflow signatures) or application-level semantics (pointer-taintedness) at runtime.

Dataflow signature checking. Our source-level semantic technique ensures that only instructions that are allowed to write to a given memory location actually do so during program execution. We achieve this by enforcing compiler-derived data dependencies at runtime using specialized programmable hardware. Because compiler analyses typically make no assumptions about the runtime layout of objects in memory, the dependencies the compiler derives are artifacts of the program's source code and not system artifacts. The technique enforces this behavior by encoding the instructions that write to the critical object as a signature and then checking the signature at runtime. Memory corruption attacks to security-critical data objects are detected because they correspond to memory access behavior nto allowed for at the source-code level.

However, an attacker could try to subvert the system by overwriting an object that the program uses to derive or influence a security-critical object's value. To prevent this, we encode the entire series of dependencies for a security-critical object as a signature for that object. This prevents the attacker from overwriting any of the objects on the dependence chain. To subvert the scheme, the attacker must modify the critical object only through instructions that *are* allowed to write to the object according to compiler-derived source-level semantics. We found that this increases the attacker's effort to mount a successful attack because it limits the number of attack points in the program and prevents the attacker from taking advantage of the object layout in memory.

Consider the example code for a password-based authentication scheme illustrated in Figure 2.

The program fragment prompts the user for a password and compares the entered password to the correct one stored in the password variable. If the user enters the correct seven-character password "`asecret`," the program outputs `Success`; if the passwords don't match, it outputs `Failed`. However, the unchecked bounds on the `gets()` function let the user enter more than seven characters, allowing other variables on the stack to be overwritten. Now assume that an attacker enters the string `attack! attack!`. This would let the attacker overwrite the password variable on the stack and falsely authenticate to the system—a classic buffer-overflow attack.

The main reason for this vulnerability is that although the programmer never implied that the password buffer should be written by the `gets()` function, the runtime system allows the write. From the source code, it seems clear that only the `gets()` function should write to the `userpass` buffer, but the runtime system doesn't enforce this—rather, it allows any instruction to write to any memory location. The attacker can easily exploit this disconnect between source-level semantics and the runtime layout of objects on the stack, and overwrite security-critical variables.

So how does the application-aware technique detect such an attack? In the last example, let's assume that both the `password` and `userpass` buffers are critical. During static compilation, the enhanced compiler identifies and encodes the set of all instructions that are allowed to write to each of the critical variables (For the sake of clarity, we consider here signatures at the granularity of program statements rather than instructions.) In this example, the only statement that's allowed to write to the password variable is `password[8] = "asecret"` on line three, and the only statement that's allowed to write to the `userpass` variable is `gets(userpass)` on line six. Suppose our signature is the set of memory addresses, so the signature for `password` is `{3}` and the signature for `userpass` is `{6}`. Now, any attempt to write to `password` from within `gets()` on line six will be detected because six isn't in the signature for the `password` variable, and the signatures won't match.

Pointer-taintedness checking. Our other technique ensures that data provided by the user isn't used as a pointer value during program execution. We've found that user data employed as a pointer value is the main cause of common memory attacks such as buffer-overflows and format-string vulnerabilities. A pointer is said to be tainted if the pointer value comes directly or indirectly from user input, thus a tainted pointer lets the user specify the target memory address to read, write, or transfer control to, which can lead to a compromise of system security. The attacker's ability to specify a malicious pointer value is crucial to the success of attacks that exploit memory corruption.

To indicate whether data is derived from user input, we extend the memory model and associate a Boolean property, called *taintedness,* with each register/memory location. The program marks any data received from external sources as tainted (external data sources can include the network, file system, keyboard, command-line arguments, and environmental variables). This is achieved through slight modifications to the operating system's user input routines. Load, store, and ALU instructions are responsible for propagating taintedness from register to register, memory to register, and register to memory. Any time the program uses a data word with tainted bytes as an address for memory access or control-flow transfer, an alert is raised and the application process is terminated. The proposed architecture is transparent to the application and requires no source-code access or compile-time type information.[5,6]

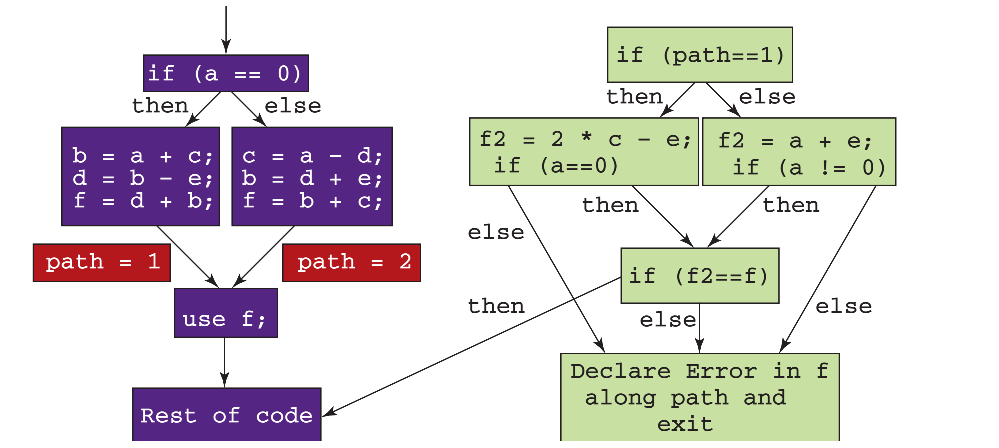

Figure 3. Error checks. The specialized software tool inserts code to track the path followed at runtime and then choose the correct check for the path. The original code is in blue, the checking code is in green, and the application instrumentation is in red.

Derivation of reliability checks

The reliability check's goal is to detect errors in the critical variable's value, which occur due to both transient hardware errors as well as software errors. One way of detecting such errors is to replicate the critical variable's computation, which usually involves considerable performance and resource costs (in terms of hardware). We propose an alternative approach: applying specialized code optimizations to form the check (that is, the sequence of instructions for recomputing the value). Our first optimization is to include only those instructions that influence the critical variable's value in generating the check (this corresponds to the backward slice of the critical variable in the program). The second optimization involves tracking the control path that the program follows at runtime and re-executing only the instructions in the slice along this path.

The reliability-check derivation technique performs the backward slicing and path specialization at compile time for each critical variable along every acyclic path in the program to derive the checking instruction sequence or expression. A specialized software tool then inserts instrumentation in the code to track the path followed at runtime and choose the correct checking expression for the path. Figure 3 illustrates this approach with a simple *if-then-else* statement. In the figure, the original code is shown in blue, and the checking code is in yellow.

Assume that *f* is a critical variable to be checked. Figure 3 shows only the instructions corresponding to the backward slice of variable *f*; this slice has two paths, corresponding to each of the two branches of the *if-then-else* statement. The specialized software tool optimizes the instructions along each path to a concise expression, which checks the critical variable's value, depending on the path followed. The expression also validates the path being followed to ensure that the path is correct.

The derived checks can be implemented in either software or hardware, but we prefer hardware because it ensures low-overhead error detection without sacrificing coverage. The hardware implemen-

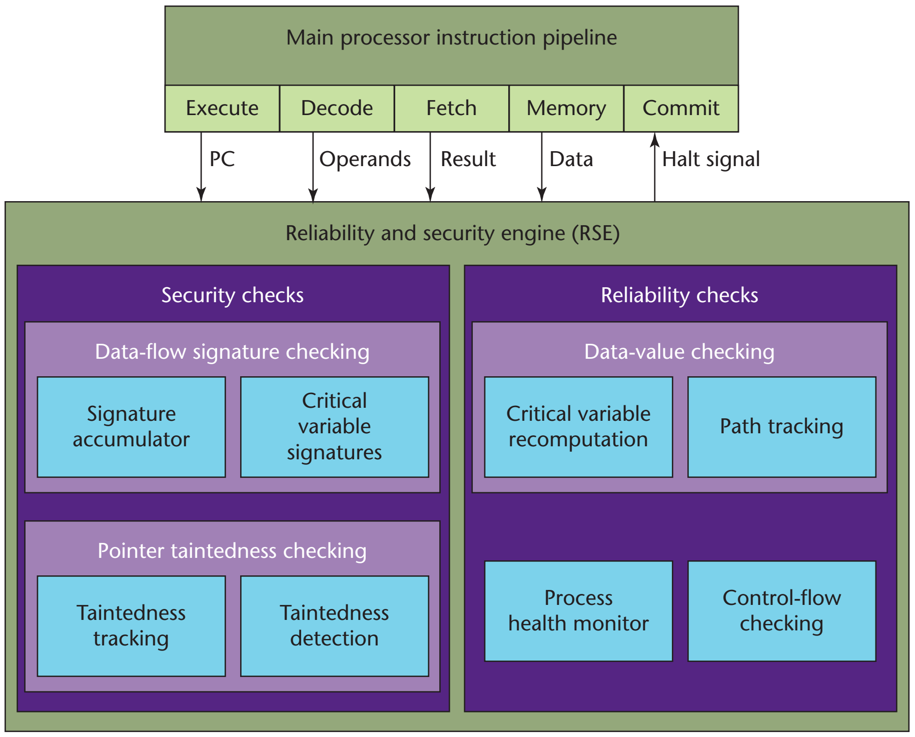

Figure 4. Framework and hardware prototype. Our reliability and security engine (RSE) is an integral part of the processor and embeds hardware modules for providing error detection and security protection.

tation of error detectors involves two steps: instrumentation of the target application with special instructions to invoke the hardware checks, and generation of an error-detection module, which is a piece of customized hardware to check the program's execution at runtime and raise an exception if one of the detectors signals an error. These two phases occur at compile time, before the application is executed, but can be executed at application load time as well. We're now leveraging the advance optimization techniques provided by the Impact compiler (http://gelato.uiuc.edu) to enhance security checks while we work to incorporate the necessary support into the compiler itself. This reliability effort in turn leverages the LLVM compiler framework.[8]

A prototype

To deploy our application-specific error-checking mechanisms, we developed the RSE, a hardware framework that's an integral part of the processor and resides on the same die. We embedded hardware modules in the RSE to provide reliability and security to the applications; the modules execute in parallel with the main processor's pipeline.[8]

We integrated the RSE prototype into the Superscalar-DLX processor, which runs on a Xilinx field-programmable gate array (FPGA). The framework offers application-specific modules tailored to provide different types of security and reliability support. Figure 4 gives a block diagram of the RSE along with the pipeline of the superscalar DLX processor. The RSE interface provides inputs for embedded hardware modules: *control-flow checking*, which detects violation of the control flow in instruction execution; *a process health monitor*, which detects operating system hangs; *data value checking*, which detects corruption of critical program variables; *dataflow signature checking*, which detects violation of data dependencies in the computation of critical variables; and *pointer-taintedness checking*, which detects memory corruption attacks in which the adversary overwrites program pointers. The RSE's configurability allows the developer to choose which modules to incorporate according to an application's requirements, thereby minimizing chip-area overhead because the unnecessary modules simply aren't included in the design.

To illustrate the RSE's security protection, we synthesized and integrated the pointer-taintedness detection module with an FPGA-based implementation of the RSE. We're currently executing realistic application scenarios with fully configured FPGA hardware and are using the same hardware base to implement dataflow-signature checking for security protection.

Although the current RSE hardware is a research prototype, our work has opened up an interesting array of implementation opportunities, including dynamic configuration of the RSE to support a system or application's changing needs, compiler extensions to enable automated generation of application-specific security and reliability checks, and trusted processing cores. To explore these opportunities, we're porting our RSE prototype to next-generation FPGA-based hardware. This transition enables experimentation with larger applications and opens avenues for future integration of RSE hardware within the Trusted ILLIAC, a configurable, application-aware, high-performance platform for trustworthy computing being developed at the University of Illinois. □

Acknowledgments

We thank our research sponsors: the US Na-

tional Science Foundation (CCR 99-02026, ACI 0121658 ITR/AP, ACI CNS-0406351, and NSF CNS 05-24695); Gigascale Systems Research Center (GSRC/MARCO); the Motorola Corporation; and the Intel Corporation. Finally, we thank our colleagues, past and present, for their contributions to the research described in this article.

References

1. M. Seecof, "AT&T 'Deeply Distressed' over Outage," *The Risks Digest*, vol. 12, no. 43, 1991, http://catless.ncl.ac.uk/Risks/12.43.html#subj2.
2. Z. Kalbarczyk, R.K. Iyer, and L. Wang, "Application Fault Tolerance with Armor Middleware," *IEEE Internet Computing*, vol. 9, no. 2, 2005, pp. 28–37.
3. K. Pattabiraman, Z. Kalbarczyk, and R.K. Iyer, "Application-Based Metrics for Strategic Placement of Detectors," *Proc. Pacific Rim Int'l Symp. Dependable Computing*, IEEE CS Press, 2005, pp. 75–82.
4. S. Chen et al., "Non-Control-Data Attacks Are Realistic Threats," *Proc. Usenix Security Symp.*, Usenix Assoc., 2005, pp. 177–191
5. S. Chen et al., "Formal Reasoning of Various Categories of Widely Exploited Security Vulnerabilities Using Pointer Taintedness Semantics," *Proc. 19th Int'l Information Security Conf.* (SEC '04), Kluwer Academic, 2004, pp. 83–100.
6. S. Chen et al., "Defeating Memory Corruption Attacks via Pointer Taintedness Detection," *Proc. IEEE Int'l Conf. Dependable Systems and Networks*, IEEE CS Press, 2005, pp. 378–387.
7. C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," *Proc. ACM Int'l Symp. Code Generation and Optimization* (CGO '04), IEEE CS Press, 2004, pp. 75–86.
8. N. Nakka et al., "An Architectural Framework for Providing Reliability and Security Support," *Proc. Int'l Conf. Dependable Systems and Networks* (DSN-04), IEEE CS Press, 2004, pp. 585–594.

***Ravishankar K. Iyer** is director of the Coordinated Science Laboratory and the George and Ann Fisher Distinguished Professor of Engineering at University of Illinois, Urbana-Champaign. His research interests include reliable and secure computing, measurement and evaluation, and automated design. Iyer has a PhD from the University of Queensland, Australia. He is an IEEE fellow, an associate fellow of the American Institute for Aeronautics and Astronautics, and a fellow of the ACM, Sigma Xi, and the IFIP Technical Committee on Fault-Tolerant Computing (WG 10.4). Contact him at iyer@crhc.uiuc.edu.*

***Zbigniew Kalbarczyk** is a research professor at the University of Illinois, Urbana-Champaign. His research interests include automated design, implementation, and evaluation of dependable and secure computing systems. Kalbarczyk has a PhD in computer science from the Bulgarian Academy of Sciences. He is a member of the IEEE, the IEEE Computer Society, and the IFIP Technical Committee on Fault-Tolerant Computing (WG 10.4). Contact him at kalbar@crhc.uiuc.edu.*

***Karthik Pattabiraman** is a PhD student at the University of Illinois, Urbana-Champaign. His research interests include application-level reliability and security, compiler analysis, and runtime system design. Pattabiraman has an MS in computer science from the University of Illinois, Urbana-Champaign. Contact him at pattabir@uiuc.edu.*

***William Healey** is a masters student at the University of Illinois. His research interests include application-level security and reliability, network security, and intrusion detection. Healey has a BS from the University of Illinois. Contact him at whealey@uiuc.edu.*

***Wen Wen-mei W. Hwu** is the Sanders-AMD Endowed Chair in the Department of Electrical and Computer Engineering at the University of Illinois, Urbana-Champaign. He also directs the IMPACT research group (www.crhc.uiuc.edu/Impact). His research interests include architecture, implementation, and software for high-performance computer systems. Hwu has a PhD in computer science from the University of California, Berkeley. He is a fellow of the IEEE and the ACM. Contact him at w-hwu@uiuc.edu.*

***Peter Klemperer** is a graduate research assistant in the Department of Electrical and Computer Engineering at the University of Illinois, Urbana-Champaign. His research interests include FPGA implementation of the RSE. Klemperer has a BS in computer engineering from the University of Illinois, Urbana-Champaign. Contact him at klempere@uiuc.edu.*

***Reza Farivar** is a PhD student at the University of Illinois, working in the Center for Reliable and High-Performance Computing. His research interests include hardware-based methods to augment security and reliability of microprocessors. Contact him at farivar2@crhc.uiuc.edu.*