

Randomized Instruction Sets and Runtime Environments

Past Research and Future Directions

Instruction set randomization offers a way to combat code-injection attacks by separating code from data (specifically, by randomizing legitimate code's execution environment). The author describes the motivation behind this approach and two application environments.



ANGELOS D.
KEROMYTIS
Columbia
University

Code-injection attacks are one of the most powerful vectors for compromising a system remotely. Attackers insert code of their choosing into a remote system and somehow induce its execution. This injected code then acts as a “beach head” through which, if undetected or otherwise unchecked, attackers can explore and use the system to their own ends. Although the remote insertion of new code into a target system can take many forms, the term *code injection* typically means that the code was surreptitiously added to an existing, running process or application (as opposed to, for example, a malicious executable received as an email attachment).

For many years, the most common method for code injection was via buffer overflow vulnerabilities. By exploiting weaknesses involving input validation and array-bounds-checking in C/C++ programs, an attacker could inject code to a remote process's address space and cause the program to cede control to the injected code. In the simplest case, the return pointer of a specific function's stack frame is made to point to the injected code, causing the program to jump to the attack code upon returning from that function. More recently, different types of code-injection attacks have also started to appear, but they typically operate at a different level of abstraction and exploit completely different vulnerabilities. SQL-injection attacks, for example, involve inserting database commands into data sent to Web applications, allowing the attacker to extract or manipulate information in a Web site's back-end database. Cross-site scripting (XSS) attacks let intruders bypass modern Web browsers' security mechanisms by making their JavaScript code appear as

if it were coming from a different, possibly trusted, site.

Researchers and practitioners have proposed several techniques to counter code-injection attacks, including safe languages, static code analysis tools, software hardening techniques, hardware extensions such as the No-eXecute (NX) feature in modern processors, attack detection and containment mechanisms, and so forth. One such technique is instruction set randomization (ISR). The basic idea behind this approach is that attackers don't know the language “spoken” by the runtime environment on which an application runs, so a code-injection attack will ultimately fail because the foreign code, however injected, is written in a different language. In contrast to other defense mechanisms, we can apply ISR against any type of code-injection attack, in any environment. Moreover, its use results in diversifying the runtime environment such that a successful attack against one process or host won't succeed verbatim against another. This is particularly useful in the context of self-propagating malware (such as worms), which depends on exploiting the same vulnerability in the same way across different systems, to compromise large numbers of systems.¹

Naturally, we can't depend on the secrecy of the language or runtime environment for any significant time period in the presence of a determined attacker. Instead, following modern cryptography's lead, we should depend on robust algorithms for creating numerous different languages or runtime environments and then choose randomly from among them. Think of this random choice as a key: we can use it to

transform legitimate, authorized code to something compatible with the corresponding instance of the runtime environment or language.

In the remainder of this article, I discuss two specific applications of ISR—protecting against binary code injection and SQL injection. I also discuss the use of ISR as an adaptive protection mechanism in a host-based intrusion prevention system. My goal is to cover the technique’s limitations—for example, it doesn’t work well with self-modifying code and requires additional debugger support—and, where possible, how to overcome these obstacles in future work.

ISR for Binaries

The first application of ISR targeted code-injection attacks in program binaries, with two independent research groups (University of New Mexico and Columbia University) demonstrating the concept and their differing implementations at the same 2003 conference.^{2,3} At the time, hardware features that enforce separation of code and data at the page level (such as the NX extension) weren’t available. Today, such features largely obviate the need for binary ISR in desktop computers and servers, but not all processors and operating systems support sophisticated memory management (a necessary component for using hardware protection features), especially in the embedded systems space.

Although the typical injection vehicle is via a buffer overflow attack, ISR itself is agnostic with respect to said vehicle. To demonstrate the concept, my team at Columbia University’s Network Security Laboratory in collaboration with Vassilis Prevelakis (Drexel University) applied ISR to network server applications because such systems generally represent often-targeted (and thus high-risk) environments. As Figure 1 shows, our approach aims to create a “randomized CPU” on which software that has undergone a compatible transformation can operate; foreign code that’s incompatible with the randomized CPU will malfunction. Depending on the underlying mechanism for implementing ISR, the CPU could execute the entire system (including the operating system kernel) in randomized mode; alternatively, the system could execute only individual processes in randomized mode, possibly using different keys for different processes. During the code randomization process, the user or administrator chooses the key at random and provides it to the CPU either at system startup or process-start time. In the latter case, the key could change periodically or even across individual invocations of the same program binary—but every time the key changes, so must the program binary. In principle, it’s even possible for the operating system to re-randomize the text segment of a running process periodically and then reconfigure the CPU accordingly.

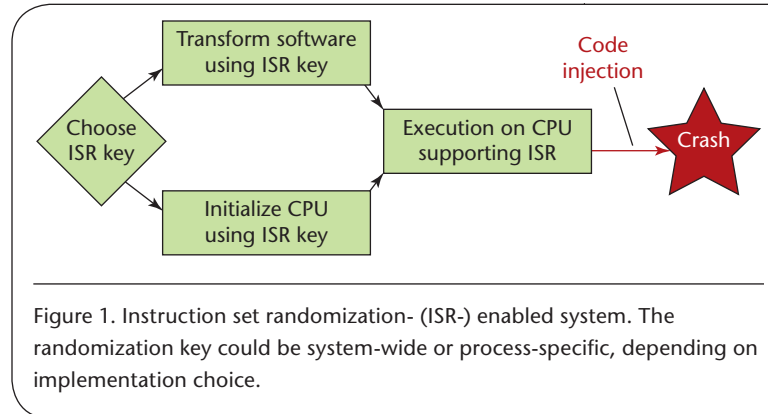


Figure 1. Instruction set randomization- (ISR-) enabled system. The randomization key could be system-wide or process-specific, depending on implementation choice.

From a security perspective, it’s desirable to change the key frequently because we’ve seen that, in some situations, attackers can use a series of carefully constructed timing attacks to guess it in linear time if it doesn’t change in response to a software failure.^{4,5} It’s also desirable (but not crucial) to use independent keys for different processes. One complication is the use of dynamically loaded shared libraries in modern systems: we can’t pre-randomize them, nor can we randomize them with a single key (several randomized processes might actively use them simultaneously). Thus, we must either use whole-system randomization (with the operating system kernel, shared libraries, and all programs using the same key) or resort to statically linking the libraries of those processes that we want to randomize, which are typically network-facing server applications.

The second complication with our scheme is that no commercially available CPU supports ISR. Although the logic for implementing ISR is relatively straightforward (requiring an additional register on which to load the ISR key and appropriate decoding logic in the CPU’s instruction decode stage), it’s impossible to retrofit existing CPUs without resorting to custom fabrication. One possibility is to implement the CPU itself inside a field-programmable gate array (FPGA). Other possibilities we briefly investigated but ultimately couldn’t pursue due to lack of resources and relevant documentation were the use of microcode updates to CPUs that support them and reprogrammable CPUs such as the TransMeta Crusoe chip. However, microcode details and update procedures are valuable assets to processor manufacturers, making it very unlikely that we would be able to implement ISR this way.

Although we could imagine several different randomization schemes, we decided to use relatively straightforward transformations that we could in principle efficiently implement in hardware. Perhaps the two most obvious involve XOR-ing the program text with the randomization key or using the key to transpose the bits within an instruction. For concreteness,

we focus on XOR as the randomization algorithm. The disadvantage of using such simple randomization schemes is that an adversary that managed to somehow see the randomized code could easily determine

The disadvantage of using such simple randomization schemes is that an adversary that managed to somehow see the randomized code could easily determine the key.

the key. Because our threat model focuses on remote code injection (that is, we aren't concerned about users with legitimate local access to the system), this was an acceptable limitation.

However, partly due to the simplicity of the randomization scheme, we needed to ensure that an adversary couldn't overcome ISR simply by exhaustively searching the key space. Although attacks against ISR are generally orders of magnitude slower than equivalent attacks against passwords or cryptographic keys because they involve interaction with a complex system over a network, it's important to minimize the probability of an adversary correctly guessing the key at random. One related complication arose from our choice of demonstration platform/processor—specifically, we chose to demonstrate ISR on the commonly available and widely used x86 family of CPUs. In contrast to reduced instruction set computer (RISC) architecture processors, which have fixed-length instructions (typically 32 or 64 bits), x86 processors use variable-length instructions. Because x86 instructions need not be aligned to any byte multiple (as almost all RISC processors require), and some instructions are 1 byte long, we would need to perform full code disassembly to correctly apply the randomization algorithm at the instruction boundary. This task is difficult even in the best of circumstances—the code might be reachable only through jump tables, object methods, or other instances of function pointers, all of which are unknown to us because we don't assume access to the source code. We compromised by using 16-bit randomization keys (which give an adversary a 1-in-65,536 chance of correctly guessing the key) and relying on the fact that most commonly used x86 compilers (including GCC and Microsoft's Visual Studio C++) seem to align code blocks to 16-bit (2-byte) boundaries by silently adding 1-byte NOP (no operation) instructions as needed. If the developer or user compiles the code with the appropriate flags, we could also use larger randomization keys (at the expense of some memory overhead). In practice, 16-bit ISR keys (with rekeying on program startup) should provide sufficient protection for most environments.

To demonstrate the feasibility (if not immediate deployability) of ISR, we ran a prototype based on emulation. To avoid the complexity of dealing with shared libraries, we decided to pursue whole-system randomization; likewise, the University of New Mexico ISR prototype pursued independent-process randomization through emulation as well, but used static linking of libraries. In our prototype, we modified the Bochs open source whole-system emulator to provide an additional register for the ISR key and the necessary logic in the instruction decode stage. We also modified CPU interrupt-handling logic to save and reload the ISR key from the stack. Thus, although our implementation used whole-system randomization, we could implement single-process randomization (and independent keys for different ISR-protected processes) under the supervision of the operating system, which simply needs to save and reload the ISR register at each context switch.

Our implementation was relatively straightforward and operated as expected—that is, it caught all the code-injection attacks we launched against it. Because this was a pure software implementation, the underlying system's performance significantly lagged behind that of a real system. Specifically, for I/O-heavy tasks such as file copying, we observed an overhead of approximately 30 percent; for more CPU-heavy tasks such as email handling, the overhead rose to 2,000 percent.

We concluded that absent hardware support or significant optimizations in the emulation method,⁶ ISR for binaries might be too expensive for wholesale use. Moreover, ISR doesn't protect against all control-hijacking attacks—for example, it doesn't protect against “jump into libc” attacks, which abuse existing program code to achieve an attacker's goals. To counter such attacks, we can use address space layout randomization (ASLR),⁷ which most operating systems today already incorporate (including Vista, Linux, and Mac OS X). ASLR also protects against code-injection attacks, so it would appear that ISR is redundant in this case. However, the current generation of 32-bit processors provides insufficient protection against determined attackers due to the (relatively) limited address space in which randomization must take place;⁸ ASLR is much more effective in 64-bit processors. Unfortunately, embedded systems can't effectively use ASLR for the same reasons as for hardware-enforced protection features. Finally, as mentioned earlier, we must be careful to change keys frequently (especially after each software failure or crash) to avoid certain key-guessing attacks.^{4,5}

An attractive feature of ISR is that it provides a “halt on failure” protection mechanism: once the injected code starts executing, it quickly terminates

the software.⁹ Furthermore, it's relatively easy to turn ISR on and off for a given process—for example, by keeping two copies of the program text and actively managing the ISR register. Similarly, we can activate ISR for selected parts of the program—that is, parts of the program's code and processes will execute in a randomized context, with the rest executing “in the clear.” This latter intuition lets us use ISR as part of a larger, adaptive, host-based protection system despite its performance penalty.

FLIPS and Adaptive Defenses

The Feedback Learning Intrusion Prevention System (FLIPS) brings together ISR and anomaly detection to create an adaptive system that allows defenses to gradually learn what constitutes malicious input to a process.¹⁰ Anomaly detection systems, which use statistical means to summarize inputs or events of interest, typically require a training phase in which the administrator feeds the system with known-good and known-bad inputs. During this phase, the anomaly detector builds models of good and bad inputs, so during operation, we can use the anomaly detection system's output to block abnormal inputs without requiring precise attack signatures. However, the non-requirement for precise signatures also introduces uncertainty and error in the classification of inputs, which can lead to anomalous inputs (“attacks”) being classified as benign and legitimate inputs classified as anomalous—these are called the false-negative (FN) and false-positive (FP) problems, respectively. FPs adversely impact legitimate user requests and actions, whereas FNs can lead to system compromise. With some exceptions, system operators try to minimize FPs, but this typically leads to over-permissive models of normalcy, which can increase the risk of FNs (and hence successful undetected attacks).

FLIPS was the first system to combine anomaly detection and software-based ISR in a feedback loop. In FLIPS, the defenses wouldn't necessarily block inputs deemed anomalous outright; instead, they would cause the process to execute with ISR enabled. If the input represented an actual attack, it would lead to a software failure, a fact that the defense mechanism then feeds back to the anomaly detection system to improve its model of normalcy. The input itself is added to a list of known malicious inputs to be filtered (signature-based blocking). If no failure occurs while processing inputs flagged as anomalous, we indicate to the anomaly detection system that it generated an FP—again, to improve the model of normalcy—and the input can be added to a list of inputs that should be passed through (to avoid ISR next time the system encounters them). Conversely, normal inputs would cause the process to execute without ISR, although if enough resources are available (or the system load

is low), the administrator can enable ISR to randomly detect FNs. The net effect of this scheme is that FPs simply cause slower processing but no outright blockage and would thus be less noticeable and objectionable to users (and administrators). Consequently, administrators can tune the anomaly detection system conservatively to minimize FNs at the cost of higher FPs.

Because we can apply ISR selectively, we can use it as part of an adaptive defense system: once an attack is identified (whether through FLIPS or some other technique, such as a honeypot system) and localized in some region of the code, we can randomize only that part of the application (potentially down to an individual function) implicated in the attack. Specifically, we randomize the function whose stack-frame return pointer is corrupted, or where a corrupted control structure (such as an overwritten function pointer) is exploited. If the program jumps to injected code upon returning from that function, a fault will occur; if the program executes without failure past the point, we can disable ISR. To enable this mode of operation, we re-implemented our ISR-enabled emulator such that it could be called from within the program as a library function; upon return from the function, the program executes inside the emulator. To terminate emulation and switch program execution to the processor itself, we simply add a call to another function inside the emulator library. Upon return from that function, the program executes directly on the processor. Note that in both cases, the program executes within the same process and address space and has access to the same program state. The last piece, then, is a binary-rewriting tool that lets us insert function calls to the emulator library inside a program binary. In this way, we incur the cost of software-only ISR as needed. Our experiments show that the performance overhead in this scenario can be very low, potentially down to zero if the vulnerable code is seldom or never used aside from the attack.¹¹

SQLrand

A second application of ISR involves protecting against SQL-injection attacks in Web applications.¹²

An attractive feature of ISR is that it provides a “halt on failure” protection mechanism: once the injected code starts executing, it quickly terminates the software.

Such applications use input received from a client (for example, as part of filling out a Web form) to populate a SQL query template. The application then transmits

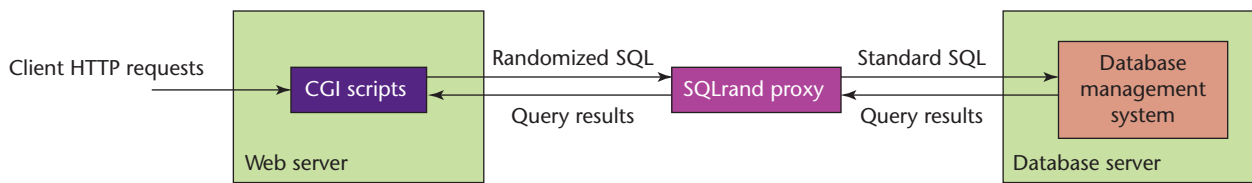


Figure 2. SQLrand system architecture. The proxy is a separate process that can run on the same or separate machines as the Web server or database management system.

the completed query to the back-end database, and the Web application processes this operation's results before presenting them to the user as part of another Web page. SQL-injection attacks exploit weaknesses in validating the input and in combining the data received from the remote user (possibly an attacker) and the SQL template—for example, consider a simple (but insecure) template populated via a cookie called USERNAME to find all orders by that user:

```
SELECT *
FROM orders
WHERE customer='USERNAME';
```

Typically, the value embedded in the cookie would be something like "ANGELOS", in which case the Web application would emit the following SQL query to the database:

```
SELECT *
FROM orders
WHERE customer='ANGELOS';
```

However, a crafty adversary could easily cause the database to return all orders by all users (exposing their private information) by editing the cookie to instead use 'or 1 = 1;' as a username, which would cause the emitted query to be

```
SELECT *
FROM orders
WHERE customer="or 1=1;";
```

More creative uses of the attack can lead to database modifications or even changes to the underlying system through stored procedures and other facilities available in modern database management systems (DBMSs). Such attacks have become extremely prevalent in recent years, surpassing buffer overflows in terms of the numbers of incidents and reported vulnerabilities in several bug-tracking databases. Despite the problem's severity, very few practical solutions exist. Surprisingly, even the research community has only recently begun looking at the problem seriously.

Our application of ISR to SQL injection, named

SQLrand, is straightforward, following our approach to binary ISR. We randomize both the underlying runtime environment (in this case, the SQL parser in the DBMS) and the SQL "program" (the template that the Web application uses). A simple approach for randomizing the SQL grammar consists of appending a random numeric tag (the randomization key) to each statement and operator in SQL. Using our previous example, the randomized SQL template using tag "123456" would look like

```
SELECT123456 *
FROM123456 orders
WHERE123456 customer=123456
'angelos';123456
```

In this case, the previously shown attack fails the parsing stage because the resulting query doesn't conform to the randomized SQL grammar. Tags can be arbitrarily long, although in practice they rarely have to be longer than 10 digits. Unlike binary ISR, an improper input will lead to a parsing failure without any random code sequence being executed. Moreover, SQL queries' looser structure and formatting requirements makes using arbitrarily long randomization keys trivial.

It's worth highlighting the difference between SQLrand and input sanitization techniques because both approaches require that the programmer identify both the "code" and the "data." With SQLrand, the programmer merely needs to randomize the code, which defeats injection attacks regardless of how the attack payload is injected. Furthermore, an improperly constructed SQLrand-enabled site or script won't work, which initial developer testing will likely catch. With sanitization, the programmer must ensure that all inputs are cleaned of potentially unsafe characters; if he or she somehow misses an execution path (a distinct possibility, given pressure to deliver functionality over security), an attack is still possible. An insufficiently sanitized script will still work, but it's difficult to enforce or verify that proper defenses are in use. Other recently proposed techniques involve "taint" tracking data received from untrustworthy sources (such as the network) as the program processes it; if the program

attempts to use such data as “code,” the system stops the operation.¹³ This can be an effective way of stopping injection attacks, but it typically requires extensive modifications to the runtime environment. Other approaches to dealing with SQL-injection problems involve domain-specific automated testing¹⁴ and static analysis techniques.¹⁵

The only problem with the SQLrand approach is that, in general, we can’t modify the SQL parser in the DBMS; this is similar to the “immutable CPU” problem in binary ISR. As in that case, we can solve the problem by introducing an intermediate processing step. As Figure 2 shows, our system (SQLrand) uses a proxy that sits between the Web application and the actual database, parses the randomized SQL queries, and emits de-randomized SQL queries to the DBMS, relaying back the results. If a parsing error occurs, the SQLrand parser drops the query.

Our implementation of SQLrand was fairly straightforward, and the prototype protected against all SQL-injection attacks with which we experimented. Furthermore, the performance impact of the randomization process and the proxy were negligible. In fact, most benchmarks failed to show a statistically significant difference in performance, while the worst reproducible result we could obtain was a 2 percent increase in query-processing latency. The reason for the minimal overhead lies in the fact that SQL parsing—and Web applications in general—already involves an interpreter or similar runtime environment that can be easily extended to support ISR.

Finally, because many runtime environments and Web applications are relatively “chatty” in case of failure (often revealing the SQL queries that failed, along with internal system variables and so forth), an attacker could induce an error report that reveals the randomization key. One straightforward solution to this problem is to parameterize the template itself, populating it with a value (or tag) received from the DBMS when the connection is first created. It’s also possible to filter Web server output such that the Web server removes instances of the tag, either by updating the filter with the specific tag in use or by using tags with a known invariant part and a random part (such as using tags starting with “SQLRANDKEY123” followed by a random sequence of digits). Care must be taken to prevent the attacker from using, as part of the attack, any variables that store the tag. Despite these caveats, we found SQLrand a powerful and easy-to-develop and use technique against SQL-injection attacks.

Future Directions for ISR

Our SQLrand work showed us that some of the most promising applications of ISR probably lie in the realm of interpreted languages. SQL injection itself remains a big problem; ways to improve the practi-

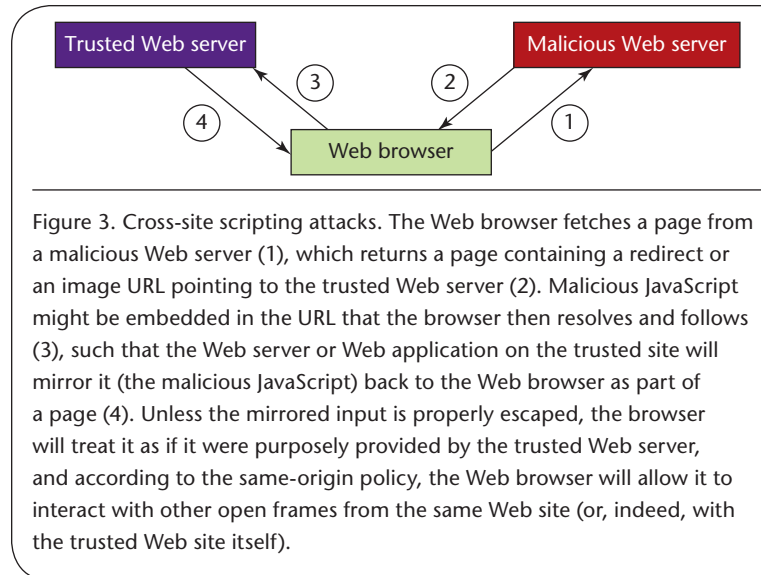


Figure 3. Cross-site scripting attacks. The Web browser fetches a page from a malicious Web server (1), which returns a page containing a redirect or an image URL pointing to the trusted Web server (2). Malicious JavaScript might be embedded in the URL that the browser then resolves and follows (3), such that the Web server or Web application on the trusted site will mirror it (the malicious JavaScript) back to the Web browser as part of a page (4). Unless the mirrored input is properly escaped, the browser will treat it as if it were purposely provided by the trusted Web server, and according to the same-origin policy, the Web browser will allow it to interact with other open frames from the same Web site (or, indeed, with the trusted Web site itself).

What Is the Same-Origin Policy?

At its core, the same-origin policy allows interaction between browser frames (through scripting) but only if the frames come from the same source domain (in most cases, this means from the same Web server). The same-origin policy prevents JavaScript code attached to a page from site A, for example, from interacting (that is, reading or manipulating the contents) with a frame containing a page from site B. Furthermore, the JavaScript code attached to a page from site A can only communicate with site A. Violation of this browser sandbox mechanism would let malicious sites manipulate user sessions with other sites and fool the user (and, in some cases, the browser) into providing login credentials for a secure site to a fake site or to a compromised Web page. To enforce the same-origin policy, browsers track from where each piece of HTML, CSS, JavaScript, Flash, and so on was received.

quality of SQLrand are thus of some importance. The primary limitation of SQLrand is that programmers must manually replace the query templates that are often embedded in their Web applications with a randomized version. Tools to help programmers with this randomization would probably also improve the chances of SQLrand’s adoption and use.

Attackers have launched strikes similar to SQL injection against various languages (including PHP and Perl); we believe that the ISR concept can be readily translated to such environments. In fact, a randomized-Perl prototype proved remarkably straightforward to construct. The only source of complexity, similar to the case of binary ISR, was the use of shared libraries (external Perl code included in a script). A good solution to this general problem remains to be found.

An interesting and relatively new problem area is that of XSS attacks against Web browsers. These attacks violate the same-origin policy browsers enforce to pro-

fect Web pages from each other (see the “What Is the Same-Origin Policy?” sidebar for a brief description). XSS attacks disguise the source of such elements (typically JavaScript code) by “bouncing” them off misconfigured or buggy servers and Web applications. This is possible if the Web server itself or a Web application on it includes in its output part of the input verbatim, as explained in Figure 3. This is a form of code injection, whereby one site inserts JavaScript (or other HTML elements) of its choosing in the contents returned from another site, such that the injected JavaScript operates with the target site’s privileges. Short of careful scrubbing of output at a Web server, it’s very difficult to protect against XSS attacks. One possibility we’re investigating involves using ISR for active content (specifically, JavaScript). Similar to the case of interpreted SQL or Perl, the JavaScript interpreter in a browser would use a different tag (ISR key) for JavaScript code received from different sites. This tag, which the Web server would append to each JavaScript keyword and operand returned by site A, would be received from the site during first contact and would remain somewhat persistent (that is, change periodically but not constantly for each user or browser). For example, the Web server could securely embed it in a cookie during the first visit to site A. The XSS-injected JavaScript wouldn’t be transformed and would thus fail to parse while executing on the browser in the context of site A (that is, under the appropriate randomization key). Several smaller complications must be resolved, including (unsurprisingly) the case of shared JavaScript code legitimately “pulled” by the browser from a third site that doesn’t implement ISR. An obvious limitation of the approach is that it requires support from both browser and server. We’re investigating ways of improving on this basic scheme.

In terms of binary ISR, several possible improvements and directions already exist. Perhaps most obviously, administrators and users can use ISR to construct secure application-specific appliances (such as hardened Web servers). If performance is a major issue, programmers can use advanced code translation techniques to significantly reduce the penalty;⁶ alternatively, they could implement the appliance via an FPGA to simulate the CPU and some of the peripherals. Programmers often use this approach to test small architectural tweaks, and several versions of Linux can run on such boards. We should be able to use our existing whole-system prototype—as is in such a runtime environment. We can also apply the general concept of randomization in different parts of the operating system/process interface.⁷

Even in a pure software implementation, ISR has some interesting uses stemming from the ability to apply it selectively. I already described its use as a targeted defense in an adaptive system, but a different use, similar in

concept to FLIPS, applies ISR as an attack sensor across a large number of identical software instances. Each instance uses ISR for only a small part of its code (thus only detecting attacks that happen to manifest within that part of the application) or for some randomly selected fraction of requests (for example, once every 10,000 requests). Consequently, the cost of monitoring for attacks is spread across many distinct, cooperating instances of the software (an “application community”).¹⁶ By adjusting the code fraction or the sampling rate, we can bring the per-instance overhead within acceptable levels but still maintain some detection (and defensive) capability. The trade-off in such a system involves performance and speed of detecting new attacks. We’re currently pursuing several research leads in this direction.

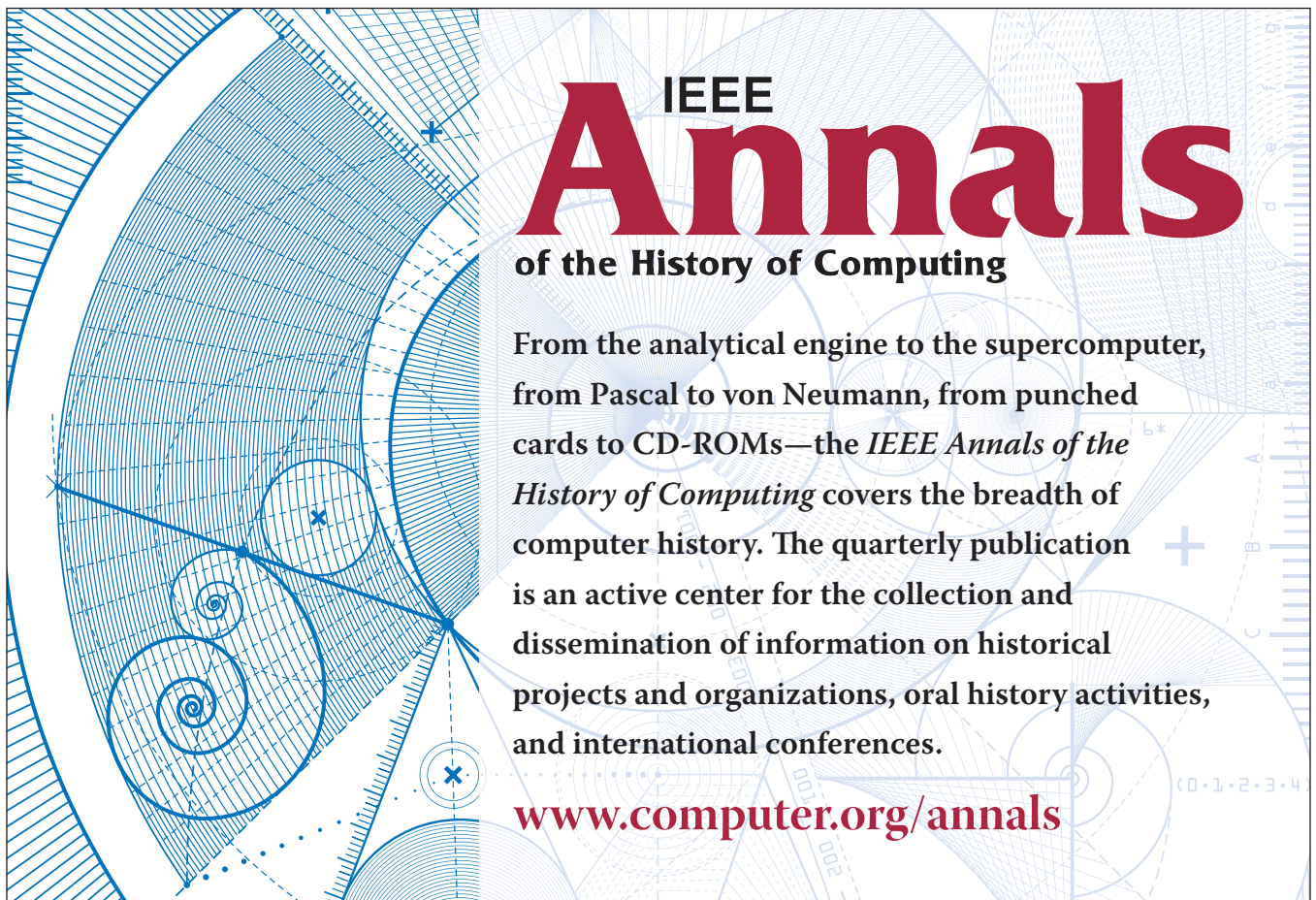
One way to view ISR is as a form of probabilistic dynamic type-checking of code;¹⁷ another is as an amendment to the classic von Neumann computer architecture and its corollaries with respect to the separation (or lack thereof) between code and data. Other techniques for providing such separation exist, all of them ultimately involving some marking scheme (of memory pages, input data, “important” addresses, and so on), but ISR is unique in that it’s applicable in so many distinct application domains. Perhaps unsurprisingly, it’s sometimes less efficient than other techniques that are tailored to specific environments or classes of vulnerabilities, and, as with any security mechanism, we must be aware of its limitations and the potential pitfalls in its use. □

References

1. S. Forrest, A. Somayaji, and D.H. Ackley, “Building Diverse Computer Systems,” *Proc. HotOS*, 1997, pp. 67–72.
2. G.S. Kc, A.D. Keromytis, and V. Prevelakis, “Countering Code-Injection Attacks with Instruction-Set Randomization,” *Proc. 10th ACM Int’l Conf. Computer and Comm. Security*, ACM Press, 2003, pp. 272–280.
3. E.G. Barrantes et al., “Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks,” *Proc. 10th ACM Int’l Conf. Computer and Comm. Security*, ACM Press, 2003, pp. 281–289.
4. A. Sovarel, D. Evans, and N. Paul, “Where’s the FEEB?: The Effectiveness of Instruction Set Randomization,” *Proc. Usenix Security Symp.*, Usenix Assoc., 2005, pp. 145–160.
5. Y. Weiss and E.G. Barrantes, “Known/Chosen Key Attacks against Software Instruction Set Randomization,” *Proc. Annual Computer Security Applications Conf. (ACSAC)*, ACSA, 2006, pp. 349–360.
6. W. Hu et al., “Secure and Practical Defense against Code-Injection Attacks Using Software Dynamic Translation,” *Proc. 2nd ACM/Usenix Int’l Conf. Virtual Execution Environments*, Usenix Assoc., 2006, pp. 2–12.

7. X. Jiang et al., "RandSys: Thwarting Code Injection Attacks with System Service Interface Randomization," *Proc. IEEE Symp. Reliable Distributed Systems*, IEEE CS Press, 2007, pp. 209–218.
8. H. Shacham et al., "On the Effectiveness of Address-Space Randomization," *Proc. 11th ACM Int'l Conf. Computer and Comm. Security*, ACM Press, 2004, pp. 298–307.
9. E.G. Barrantes et al., "Randomized Instruction Set Emulation," *ACM Trans. Information and System Security*, vol. 8, no. 1, 2005, pp. 3–40.
10. M.E. Locasto et al., "FLIPS: Hybrid Adaptive Intrusion Prevention," *Proc. 8th Int'l Symp. Recent Advances in Intrusion Detection*, Springer, 2005, pp. 82–101.
11. S. Sidiroglou et al., "Building a Reactive Immune System for Software Services," *Proc. Usenix Ann. Technical Conf.*, Usenix Assoc., 2005, pp. 149–161.
12. S.W. Boyd and A.D. Keromytis, "SQLrand: Preventing SQL Injection Attacks," *Proc. 2nd Int'l Conf. Applied Cryptography and Network Security*, Springer, 2004, pp. 292–302.
13. W. Halfond, A. Orso, and P. Manolios, "Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks," *Proc. 14th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, ACM Press, 2006, pp. 175–185.
14. M. Emmi, R. Majumdar, and K. Sen, "Dynamic Test Input Generation for Database Applications," *Proc. Int'l Symp. Software Testing and Analysis*, 2007, pp. 151–162.
15. N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)," *Proc. IEEE Symp. Security and Privacy*, IEEE CS Press, 2006, pp. 258–263.
16. M.E. Locasto, S. Sidiroglou, and A.D. Keromytis, "Software Self-Healing Using Collaborative Application Communities," *Proc. ISOC Symp. Network and Distributed Systems Security*, Internet Soc., 2006, pp. 95–106.
17. R. Pucella and F.B. Schneider, "Independence from Obfuscation: A Semantic Framework for Diversity," *Proc. Computer Security Foundations Workshop*, IEEE CS Press, 2006, pp. 230–241.

Angelos D. Keromytis is an associate professor with the Department of Computer Science at Columbia University, where he also serves as director of the Network Security Laboratory. His research interests revolve around systems and network security. Keromytis has a PhD in computer and information science from the University of Pennsylvania. He is a member of the IEEE and a senior member of the ACM. Contact him at angelos@cs.columbia.edu.



IEEE
Annals
of the History of Computing

From the analytical engine to the supercomputer, from Pascal to von Neumann, from punched cards to CD-ROMs—the *IEEE Annals of the History of Computing* covers the breadth of computer history. The quarterly publication is an active center for the collection and dissemination of information on historical projects and organizations, oral history activities, and international conferences.

www.computer.org/annals