Chaos for a Fast, Secure, and Predictable Future

John Criswell
University of Illinois
criswell@illinois.edu

Vikram Adve
University of Illinois
vadve@illinois.edu

1. INTRODUCTION

Violating a program's semantics for fun and profit is a time honored hacker tradition. Compilers defend against such fiends by inserting run-time checks to enforce semantic safety properties. Safe language compilers insert type checks for down-casts, information flow compilers [11] add run-time checks to prevent information leakage, and tools like SAFECode [8], WIT [1], and DFI [6] insert run-time checks to detect memory errors in C code. While effective, run-time checks often incur extra CPU and/or memory overhead. This is especially true when enforcing safety properties on C code [8, 1].

Instead of detecting or tolerating violations of semantic safety properties, perhaps we can gain better performance by making the results of such violations *unpredictable*. Attackers often rely on violations of semantic safety properties having specific, predictable outcomes at the machine-code level. What if violations of safety properties could be *guaranteed* to produce *unpredictable* behavior?

While randomization for security has been proposed for limited purposes [4, 5, 10, 15, 3], we believe that there is a more general principle afoot: many attacks utilizing semantic violations that are stopped using run-time checks can also be reliably thwarted using randomization techniques. The primary difference is that the former detects or prevents violations while the latter permits a violation but prevents an attacker from exploiting it.

In Section 2, we describe previous work that illustrates a semantic safety property used to enforce memory safety and how randomization can provide similar safety with less CPU and address space overhead than run-time checks. We will then generalize this principle and argue that randomization can enforce other semantic safety properties by showing novel, potential randomization techniques for enforcing information flow policies, type-safety, and data-flow integrity.

2. MEMORY SAFETY

One method of thwarting memory safety attacks is to ensure that loads and stores do not access memory objects other than those belonging to the points-to set of the pointer that the load or store dereferences. SAFECode [8] inserts run-time checks to ensure that the points-to analysis that the compiler computes is not violated at run-time; WIT [1], a similar system, enforces this safety property on stores only.

Data Space Randomization (DSR) [4, 5] also computes a points-to graph but doesn't insert run-time checks. Instead, it assigns a randomly chosen XOR encryption/decryption key to each points-to set and instruments stores and loads to encrypt/decrypt data using lightweight XOR encryption. Violation of the points-to analysis at run-time causes a load instruction to decrypt data with the wrong key; the result of the load is therefore unpredictable and useless to an attacker.

DSR addresses the same semantic safety property (points-to analysis) as SAFECode. However, it is faster than SAFECode (20% [5] vs. 27% [8] on the Olden benchmarks). Switching to a direct lookup data structure [1] in SAFECode could trade-off virtual address space for speed; DSR suffers no such tradeoff for its performance. DSR may also perform better for multi-threaded code. Unlike SAFECode, DSR has no lookup data structures which require locking for safe concurrent accesses.

Could randomization be used to enforce other semantic safety properties? We believe it can.

3. DYNAMIC INFORMATION FLOW

Information flow languages like JIF [11] allow a programmer to specify a policy for data-flow within the program by attaching labels to variables describing the confidentiality of the data stored within the variable. Some of these languages must track a label describing the confidentiality of the data because static analysis cannot guarantee that confidential information does not leak.

Consider a hypothetical programming language in which variables are references to objects. Each object has an implicit label describing the confidentiality (high or low) of the data within it. The language permits variables that can reference objects with low confidentiality data, high confidentiality data, or data of any confidentiality (denoted by a low, high, or any specifier as part of the variable's type).

Consider the following program in this language:

```
1: high int h;
2: low  int l;
2: any  int v;
3:
4: v = l;
5: if (f(...) == 5)
6:   v = h;
7: l = v;
```

A compiler may insert a run-time check before line 7 because it does not know a priori whether `v` references an object of low or high confidentiality. If line 6 is executed, the program should terminate to prevent information downgrade via reads of `l`.

Randomization provides an alternative approach. The compiler picks a set of encryption keys for low confidentiality (k_e^l/k_d^l) and high confidentiality (k_e^h/k_d^h) data. When

reading/writing an object, the appropriate keys are used to encrypt/decrypt the data. The key to use can be computed statically for variables typed as low or high.

If, at run-time, data from a high confidentiality object is read via a low-typed variable, then reads using the low-typed variable will decrypt data encrypted with k_e^h using k_d^l. The resulting value will be unpredictable, making the downgrade of information useless to an attacker.

4. TYPE SAFETY

Unsafe languages like C++ make run-time type checking inefficient because pointers may point into the middle of objects. Instead of using simple pointer arithmetic to locate a class field holding the object's type, C++ would need to use lookup structures [9, 7, 2] or steal bits from the pointer representation [13].

Randomization could alleviate lookups without trading off address space: during compilation, the compiler assigns to each field f of each type T a random encryption/decryption key pair (k_e^{Tf}/k_d^{Tf}). Loads and stores to fields of a static type T use the encryption (decryption) key of the appropriate field of the static type. Fields invisible at the source level e.g., the pointer to the virtual function table, use special encryption/decryption keys (k_e^s).

During correct execution, the program behaves normally. However, consider an execution where an incorrect downcast creates a pointer with static type A pointing to an object of type B (where B is not a subclass of A). Since there is no downcast check, loads to fields via the pointer will use the decryption key k_d^A; however, data stored into those fields will be encrypted either with k_e^B or the special encryption key k_e^s (depending upon whether class fields or compiler-inserted fields are accessed). The loads will return unpredictable results; an attacker will not be able to use the invalid type cast to cause specific behavior.

This technique also maintains type safety for dangling pointer dereferences. If the memory for the old object has not been overwritten, the correct encryption/decryption keys are used, yielding a type-safe result. If the memory is reused for an object of a different type, then the wrong decryption keys are used, and unpredictable results occur.

5. DATA FLOW INTEGRITY

Data-flow integrity (DFI) [6] inserts run-time checks into a program to ensure that a statically computed, inter-procedural reaching-definitions analysis is not violated by memory safety errors at run-time. DFI provides safety guarantees that are stronger than SAFECode's [8] and WIT's [1], but it incurs high overhead.

It may be possible to achieve a subset of DFI's safety guarantees using randomization. Like DFI, the compiler first performs reaching definitions analysis. The compiler would then create equivalence classes; each equivalence class contains all reaching definition sets whose intersection is non-empty. Each equivalence class could then be assigned an encryption/decryption key; stores and loads would then encrypt/decrypt data using the appropriate key.

If a memory error causes a load to read data encrypted by a store that does not belong to its equivalence class, then the load will decrypt the data with the wrong decryption key. The program will then exhibit undefined and unpredictable behavior. While this approach is not as strict as DFI, it should be faster; it does not require a large lookup table, and it does not need to search through a set of identifiers representing store instructions [6].

6. FUTURE CHALLENGES

Providing security via randomization appears promising, but significant research challenges remain. First, small values (16 bits or less) can be guessed by brute force [12, 14]; encrypting them may be pointless or even dangerous. Second, buffer overread attacks [14] can be used to mount known-plaintext attacks against XOR encryption; restrictions on I/O or improved encryption techniques may be needed.

The future is bright but uncertain. Perhaps, with more work, we can make it a chaotic but certain future for software safety.

7. REFERENCES

[1] Akritidis, P., Cadar, C., Raiciu, C., Costa, M., and Castro, M. Preventing memory error exploits with WIT. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 263–277.

[2] Akritidis, P., Costa, M., Castro, M., and Hand, S. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the Eighteenth Usenix Security Symposium* (August 2009).

[3] Berger, E., and Zorn, B. Diehard: Probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (2006).

[4] Bhatkar, S., and Sekar, R. Data space randomization. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (July 2008), vol. 5137/2008 of *Lecture Notes in Computer Science*, pp. 1–22.

[5] Cadar, C., Akritidis, P., Costa, M., Martin, J.-P., and Castro, M. Data randomization. Tech. Rep. MSR-TR-2008-120, Microsoft Research, September 2008.

[6] Castro, M., Costa, M., and Harris, T. Securing software by enforcing data-flow integrity. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 147–160.

[7] Dhurjati, D., and Adve, V. Backwards-compatible array bounds checking for C with very low overhead. In *International Conference on Software Engineering* (Shanghai, China, May 2006), pp. 162–171.

[8] Dhurjati, D., Kowshik, S., and Adve, V. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Canada, June 2006), pp. 144–157.

[9] Jones, R. W. M., and Kelly, P. H. J. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging* (1997), pp. 13–26.

[10] Lin, Z., Riley, R. D., and Xu, D. Polymorphing software by randomizing data structure layout. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (June 2009), vol. 5587/2009 of *Lecture Notes in Computer Science*, pp. 107–126.

[11] Sabelfeld, A., and Myers, A. Language-based information-flow security. *IEEE Journal on Selected Areas in Comm.* (2003).

[12] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *Proceedings ACM Conf. on Computer and Communications Security (CCS '04)* (2004), pp. 298–307.

[13] Steenkiste, P., and Hennessy, J. Tags and type checking in lisp: hardware and software approaches. *SIGPLAN Not. 22*, 10 (1987), 50–59.

[14] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., and Walter, T. Breaking the memory secrecy assumption. In *EUROSEC '09: Proceedings of the Second European Workshop on System Security* (New York, NY, USA, 2009), ACM, pp. 1–8.

[15] Team, T. P. aslr.txt. http://pax.grsecurity.net/docs/aslr.txt.