An Overview of Security-Driven Compilation*

Rahul Simha†, Alok Choudhary‡, Bhagi Narahari§, Joseph Zambreno¶

November 4, 2004

Abstract

The growing importance attached to the security of software systems has focused considerable attention on various problems related to security in computing. A vast array of research efforts ranging from highly technical cryptographic techniques all the way to higher-level information policy are currently engaged in addressing the security needs of the next generation of computing infrastructure. This paper reviews the role that compilers and compiler research can play in this realm. Because they routinely extract useful program structure and transform code, compilers are uniquely positioned in some ways to have a direct impact in the creation of more secure software. These features, among others, together with a fresh approach towards software protection based on compilation, can result in more robust applications and computing infrastructures. This paper will review on-going efforts, explore future research directions and make the case for renewed attention in this important area at the intersection of compiler and security research.

†Department of Computer Science, The George Washington University, Washington, DC 20052. Email: `simha@gwu.edu`

‡Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL. Email: `choudhar@ece.northwestern.edu`

§Department of Computer Science, The George Washington University, Washington, DC 20052. Email: `narahari@gwu.edu`

¶Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL. Email: `zambro1@ece.northwestern.edu`

1 Introduction

Research into *software security* or *software protection* has intensified in recent years following several high-profile disruptions of computing systems. Hackers all over the world know that the key steps to attacking a software system is to first *understand* the software, and then to *tamper* with the software to enable a variety of full-blown attacks. The growing area of software protection aims to address the problems of *code understanding* and *code tampering* along with related problems such as *authorization*. The general objective of research efforts in this area is to provide techniques to help proper authorization of users, to prevent code from being tampered with and to also make it harder for attackers to extract information that could be used in identifying system vulnerabilities. Computer security has witnessed several decades of research that has produced techniques in a variety of security-related areas including among others: cryptography, security protocols, proof systems, intrusion detection, authentication, policy definition and enforcement, secure communication, and architectural support for secure computing. Over time, security objectives have invited cross-disciplinary interest as researchers in theory, networking, architecture, data mining and signal processing, have brought their expertise towards addressing problems in security.

The purpose of this paper is to review research that addresses the following question: what role can compilers and compiler research play in achieving security objectives? By reviewing current efforts in this area, and outlining potential directions for future research, we make the case that the research community can benefit from a greater level of attention directed at the intersection of compilers and security. Just as compiler and language research has evolved

over the years to encompass objectives such as parallelism, interpreted languages, virtual machines, architecture and hardware description languages, so too, we argue, should compilers take up security as a primary objective.

The belief that that compilers should play a stronger role in building future secure systems is based on the following rationale: First, compilers are well-positioned in their "gateway" role in software creation: almost all executable software today is created with compilers, and therefore, effective techniques embedded in a compiler can automatically reach executables without conscious effort on the part of developers. Second, compilers already extract a great deal of program structure, enabling a more detailed automated analysis than may be possible with pre or post-compilation tools. Third, compilers modify code in various ways; these transformations can be used for many security objectives. In fact, this key feature is part of the foundation of many existing software protection approaches. Fourth, because decades of research in compiler optimization have made their way into compiler infrastructures, the compiler is probably the best place to manage security-performance trade-offs. Fifth, many well-known compilers such as gcc centralize architectural information in one place and therefore, security transformations that are implemented at the level of intermediate code or higher can immediately impact code written for a variety of processors. Sixth, compilers often exploit architectural features such as instruction-level parallelism and are therefore well-positioned to exploit similar hardware support for security. Finally, and perhaps most importantly, because compiler research involves managing entire compiler infrastructures, compiler researchers tend to be experienced in all aspects of compilers, ranging from language constructs at the front-end to low-level hardware features at the back-end. This encourages a comprehensive top-down view in supporting security objectives.

In this paper, we review on-going efforts in many sub-areas of compiler-security research. These include high-level language design issues as well as low-level hardware issues. Because compiler research includes this broad span of issues, there are equally many promising directions for future research.

To help frame our discussion of issues, we divide attacks on software integrity into four kinds. In a *code tampering attack*, executable code is tampered with for a variety of purposes including interfering with licensing, changing the feature-set and using the code to attack other devices. A typical example is the well-known buffer-overflow attack on the networking stack. An *authorization attack* is when code tampering is used to circumvent a vendor's checking of permission in executing software on a particular hardware. A *data tampering attack* refers to modifications of data such as passwords or internet addresses to enable hackers to exert control of software behavior. Finally, the term *code understanding* is self-explanatory: it is the necessary first step to any of the other attacks.

The remainder of this paper is organized as follows. In Section 2, we describe past and on-going work in the field of software protection, categorizing them in various areas that we hope will be helpful to the reader. We follow that in Section 3 by describing a few projects in a little more detail, especially focusing on projects that we are most familiar with. Finally, Section 4 speculates on future directions.

2 Related Areas of Research

In this section, we review past and current work in the general area of software security as related to compilers. A broad survey of several software protection techniques appears in [25]; these range from tamper resistant packaging to hardware copyright notices. A survey of the broader area of software security in the context of DRM appears in [10, 62]. Note that several companies have formed the so-called Trusted Computing Platform Alliance (TPCA) to provide hardware-software solutions for software protection [69]. Finally, an excellent survey of tools for open-source software is provided in [16], with a summary of analysis tools (some of which involve source code parsing) as well as runtime tools (such as StackGuard) that can be broadly considered in the area of languages and compilers. Our review will focus on projects that also feature code generation, and deal primarily with the compiler tool chain. It should

be noted that the system level tools [16] are orthogonal to what we discuss in this review and therefore can be used on top of the techniques that we discuss in this review.

2.1 Copyright notice and Watermarking

The oldest "prevention" technique is to embed a copyright notice into the code and to check for the existence of the copyright upon execution. Because an ordinary insertion of text is easy to find, compiler techniques can be used to generate a number of ingenious watermarks by transforming the code. For example, the arrangement pattern of basic blocks or the order of functions in memory can be used to encode a unique pattern that is hard to discern. A survey and taxonomy of watermarking techniques can be found in [13, 14].

Watermarks are very useful in tracking and identifying code that may be illegally copied because unique watermarks can be used to trace the original owner. However, they cannot prevent attacks during execution unless accompanied by code that periodically checks for the presence of the watermarks. Note also that this form of software protection is probably the easiest to hack: the code that checks is simply removed or routed around. Also, a copyright check even if valid does not prevent a hacker from actually modifying the code for unauthorized purposes.

2.2 Obfuscating Compilers

In this technique, now receiving much attention, code is deliberately mangled while maintaining correctness to make understanding difficult – a survey of obfuscation techniques appears in [12]. Obfuscation techniques range from simple encoding of constants to more complex ones that re-arrange or transform code [11, 12]. Techniques in [57, 58] also propose transformations to the code that make it difficult to determine the control flow graph of the program, and show that determining the control flow graph of the transformed code is NP-hard. Theoretical limitations are discussed in [6]. Obfuscation is attractive because it is simple to use, does not involve any keys and addresses the code understanding problem. It

can be used in conjunction with other techniques that address code tampering. Obfuscation can also simultaneously provide a watermark. However, many obfuscation techniques can be attacked by designing tools that automatically look for obfuscations. Another approach to attacking obfuscation techniques is to run the code in a debugger and to identify vulnerabilities by stepping through the code in the debugger.

2.3 Compiler-Generated Signatures

A *digital signature* [51] is a standard cryptographic technique to help identify whether a block of text has been modified. Typically, a text is hashed to form a digest and the digest is signed with a key. Then, any attack that modifies the text can be detected since the modified text will result in a different digest with extremely high probability. Often, the hash is simply a checksum, an idea that has been exploited for computing digests of executable code.

The work in [8] introduces the concept of *guards*, pieces of executable code that perform checksums to guard against tampering. In [29], the authors propose a dynamic self-checking technique to improve tamper resistance. The technique consists of a collection of "testers" that test for changes in the executable code as it is running and reports modifications. A tester computes a hash of a contiguous section of the code region and compares the computed hash value to the correct value. An incorrect value triggers the response mechanism. They note that the performance is invariant until the code size being tested exceeds the size of the L2 cache, and a marked deterioration in performance was observed when this occured. The techniques in [8, 29] are compatible with copy-specific watermarking and other tamper resistant techniques and requires using these to provide a high level of software protection. In [4] a self-checking technique is presented in which embedded code segments verify integrity of the program during runtime. The above proposed "self-checking" approaches essentially compute checksums on code to assert code integrity [8, 29]. This computation is exactly the same as any other digest or MAC computation for secure communication: it relies on the high probability that a modification to the code will create a modified checksum.

3

Such digest checking is attractive because digests can detect any kind of modification to code or data, and is relatively inexpensive in terms of computation effort. Furthermore, no keys are required if the digests are well-hidden. At the same time, these techniques strongly rely on the security of the checksum computation itself. If these checksum computations are discovered by the attacker, they are easily disabled. However, in many system architectures, it is relatively easy to build an automated tool to reveal such checksum-computations. For example, a control-flow graph separates instructions from data even when data is interspersed with instructions; then, checksum computations can be identified by finding code that operates on code (using instructions as data). This problem is acknowledged but not addressed in [29].

2.4 Static Analysis and Runtime Support

Static code analyzers scan source code and alert the programmer about problems that might be exploited for attacks. An excellent survey of tools, both static analyzers as well as those with runtime complements, is provided in [16]. Static analysis tools such as BOON [56] and CQual [26] scan C source code to find potential buffer overflows or inconsistent usage of values. MOPS [9] uses a finite state machine model of of what is considered valid behavior for a particular program. If a property is violated during analysis, *i.e.,* an illegal state is reached in the finite-state automaton, the programmer is alerted about a potential vulnerability.

Static analysis is often complemented by runtime support. This idea is not new – languages that enforce array bound checking have existed since the early days of computing. However, what is new is that these ideas can be carried over into languages like C with the help of tools such as StackGuard [55] (to ensure that return addresses are valid) and FormatGuard [17] (to address attacks on C's printf function). While static and runtime analysis are focused on particular types of attacks, the general idea of ensuring correct behavior can be made stronger by guaranteeing properties, as we discuss next.

2.5 Proof Carrying Code

Proof-Carrying Code (PCC) is a technique by which a host can verify code from an untrusted source [38, 40, 41, 2, 5, 7]. Safety rules, as part of a theorem-proving technique, are used on the host as guarantees for proper program behavior. Applications include browser code (applets) [5] and even operating systems [38]. PCC is attractive for several reasons. One advantage of proof-carrying software is that the programs are self-certifying, independent of encryption or obscurity. A second advantage is that one can formally assert properties and state guarantees. There is ongoing work on providing a compiler that can build proofs automatically during compilation. The PCC method is essentially a self-checking mechanism and is vulnerable to the same problems that arise with the code checksum methods discussed earlier; in addition they are static methods and do not address changes to the code after instantiation.

2.6 Programming Languages

A compelling approach to incorporating security into software is to force the issue at the early development stages, into a programming language itself or in tools that rewrite programs to achieve security objectives. Several efforts [31, 34, 39, 45] have taken up this approach, resulting in techniques that use type-safe assembly [37], language modification [31, 39] or reference monitors [23, 45]. Comprehensive surveys of the general approach can be found in [34, 45].

A reference monitor is a piece of software, sometimes with hardware support, that observes the execution of a program to see whether memory bounds or forbidden accesses are violated. Such monitoring capability may be incorporated into a trusted operating system or may be "in-lined" into existing compiled code [23]. The paradigm of typed intermediate languages [27, 28] provides for type information to be maintained and verified through the backend of the compilation process [28, 37]. A type-checker then verifies type safety, allowing programmer errors as well as tool-chain errors to be caught. This approach, similar to the proof-carrying code concept described earlier, is taken a step further in the notion of a certifying compiler. For example, the Java com-

piler can be said to certify that it produces type-safe bytecode, some properties of which are checked at load time by the JVM. Language-based approaches also include formal language methods, such as the extension of pi-calculus for security [1].

These approaches form a useful complement to the compiler-based approaches described thus far. Language-based approaches go a step further in security by attacking the general problem of correctness and incorporating safeguards against programmer errors. At the same time, direct modifications to a language require a buy-in from the community and may take time to find their way into standards. The long-term impact of language-based approaches is that useful features often find acceptance in the next generation of languages, as Java has shown.

2.7 Compiling for Cryptographic Architectures

We now describe some custom hardware approaches to software protection because their relevance in terms of compilers is that code generation is usually followed by encryption, a feature that can be considered a back-end compiler function.

In a secure coprocessor, programs can be run in an encrypted form on these devices thus never revealing the code in untrusted main memory memory and thereby making it difficult to understand or tamper with. A number of secure coprocessing solutions have been designed and proposed, including systems such as IBM's Citadel[43], Dyad [53, 63, 64], the Abyss and μAbyss systems [61, 60, 59], and the commercially available IBM 4758 which meets the FIPS 140-1 Level 4 validation [30, 49, 50]. An alternative to a co-processor is a processor that is designed to directly execute encrypted code, such as the architecture proposed in [35], in which an execute-only memory (XOM) allows instructions stored in memory to be executed but not manipulated.

Like a custom processor, a custom operating system on a standard processor presents an alternative approach to software security. However, they are expensive to build and maintain. A more promising approach might be an operating system such as SE-Linux [36] with features that enable customizable se-

curity.

FPGA's have been used for security applications mainly for the purpose of speeding up cryptographic computations [19, 44, 52, 33]. The FPGA manufacturer Actel [15] offers commercial IP cores for implementations of the DES, 3DES, and AES cryptographic algorithms and is currently developing new anti-fuse technologies that would make FPGAs more difficult to reverse-engineer [20].

2.8 Joint Compiler-Hardware Approaches

The hardware solutions described above that operate on fully-encrypted executables are attractive because of the quality of security they are able to provide. At the same time, they require a substantial buy-in from hardware manufacturers and can considerably slow-down execution speed.

Such a highly-secure hardware approach and the obfuscation techniques described earlier together form two ends of a security-performance spectrum. At one end are the hardware schemes that are either slower or require special purpose processors, and at the other end are obfuscation like schemes that are efficient but provide limited security. These extremes invite an approach that allows system designers to position themselves where they choose on the security-performance spectrum.

In [65, 66] we describe a joint compiler-hardware approach that is designed to allow systems to be positioned at various points in the spectrum. The hardware used is an FPGA (Field Programmable Gate Array), which offers field-programmability and is readily available with many processor cores. This approach is described next in Section 3.2.

3 Some Current Projects

3.1 HIDE - Hardware Assisted Bus-Leakage Protection

When memory and processor communicate across an open bus, it is relatively easy to record the bus traffic in order to extract information, even if the data is encrypted. When memory and processor are on

5

the same board, a sophisticated attacker with access to modern electronic laboratory equipment can insert probes to record the traffic between memory and processor for the same purpose.

As shown in [67, 68], applications have specific execution patterns that are easily identifiable from the trace of memory addresses that bus snooping provides. The HIDE project [67, 68] describes an approach to obfuscate these access patterns by changing the locations of data during program execution. Their approach is to randomly permute data addresses so that a particular piece of data is regularly moved to new locations, and accessed from the new locations in order to obfuscate the true access pattern. They show that a modest addition to hardware can implement the permutation efficiently enough to impose a minimal overhead on overall execution time.

3.2 SAFEOPS - A Joint Compiler/Hardware Approach

SAFEOPS (Software/Architecture Framework for the Efficient Operation of Protected Software) [65, 66] is a compiler/FPGA technique in which the processor is supplemented with an FPGA-based *secure hardware component* that is capable of fast decryption and, more importantly, capable of recognizing and certifying strings of "codes" (keys) hidden in regular unencrypted instructions - see Figure 1. Parts of executable code are either fully encrypted with standard private-key techniques [18] or contain embedded keys. The fully encrypted segments are decrypted by the FPGA. The segments containing embedded keys are processed by the FPGA before they reach the L2 cache, allowing the FPGA to examine the veracity of the keys. For example Figure 1 shows that the first part of the executable is encrypted whereas the second part of the executable shows a hidden code A and, at a distance d from A, an instruction A'. Upon recognizing A, the FPGA will expect $A' = f(A)$ at distance d (where f is computed inside the FPGA); if the executable is tampered with, this match is highly unlikely to occur and the FPGA will halt the processor. The code sequences are hidden within both within instructions and data.

One way to embed codes through compilation is to use the register allocator, since register assignment presents some degrees of freedom. Consider a sequence of instructions comprising the instruction stream that use registers. If we focus on, say, the first register in each register-based instruction, the resulting sequence of registers so used in the instruction stream is called the *register stream*.

The key observation is that this register stream is determined by the register allocation module of the compiler. In the FPGA hardware, the register stream can extracted from the instruction stream, in addition to other information such as particular opcodes of interest in the embedding mechanism.

Suppose we use register R_1 to encode '0' and R_2 to encode '1.' Then, the particular sequence of registers $R_1, R_2, R_2, R_1, R_1, R_2$ corresponds to the key **0 1 1 0 0 1**. This key is then compared against a cryptographic function of the opcode stream in the FPGA.

Thus, in the general technique, the compiler performs instruction filtering to decide which instructions in opcode stream will be used for checking. The compiler then uses the flexibility of register allocation to bury a key sequence in the register stream. Upon execution, the entire instruction stream is piped through the secure FPGA component, which is set up to recognize the particular register stream of interest. The FPGA then extracts both the filtered opcodes and the register sequences for comparisons. If a violation is detected, the FPGA halts the processor. If the code has been tampered with, there is a very high probability that the register sequence will be destroyed or that the opcode filtering will pick out a different instruction.

As described in [65, 66], the technique can be applied to in its simplest form to small embedded processors with FPGA cores that have little capability for full encryption. More importantly, the register allocation is a proper register allocation so that the executables can run on processors that don't feature a policing FPGA. Finally, by using private keys in the FPGA, a higher level of security can be achieved by using the private key against the stream along with cryptographic hashing. These techniques together with the compiler's ability to extract program structure and perform register alloca-
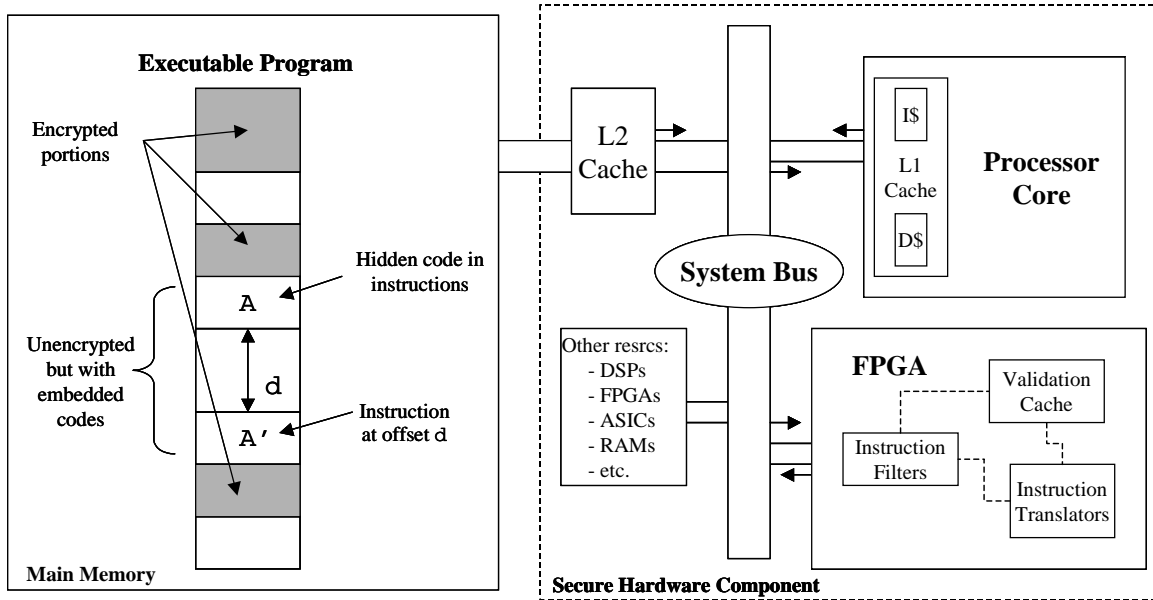
6

Figure 1: Conceptual view

3.3 CODESSEAL - Compiler/FPGA Approach to Secure Applications with Encrypted Execution and Data

CODESSEAL (COmpiler DEvelopment Suite for SEcure AppLications) [42] is a project focused on joint compiler/hardware techniques for fully encrypted execution, in which the program and data are always in encrypted form in memory. Just because executables are encrypted in main memory during execution doesn't mean attacks cannot be effective. Several types of replay, data and structural attacks, such as control flow attacks, are possible that a resourceful attacker can use to systematically uncover particular behaviors. We term such attacks as EED attacks – attacks on *Encrypted Executables and Data*. To help detect such attacks, compilers will need to play a key role in extracting structural information for use by supporting hardware.

EED attacks are attacks on highly-encrypted systems from a resourceful adversary who may not need to decrypt. The attacks are based on exploiting structure in encrypted instruction streams and data that can be uncovered by direct manipulation of hardware in a well-equipped laboratory. For example, consider a resourceful attacker who is able to carefully take apart captured hardware, has access to hardware manuals and data, and has modern laboratory equipment to directly manipulate the hardware. Such manipulation includes controlling the bus, re-writing memory and even supplanting the processor with a malicious variant. We use the term *structural integrity* to refer to the proper execution path of a program when the data is assumed to be correct. Since an EED attacker can alter the control flow without decryption or even touching the data, we refer to such an attack as a *structural EED attack* (also called *control flow attacks* in the literature). A second kind of attack arises from considering EED attacks on data integrity. There are two sub-categories of interest here, the regular data used by an application and the runtime data (stack, heap) needed for proper execution.

Our technique involves the use of an FPGA placed between main memory and the cache that its closest to main memory (L1 or L2, depending on the system) – see Figure 2. The instructions and data are loaded into the FPGA in blocks and decrypted by keys that are exclusive to the FPGA. Thus, the decrypted code
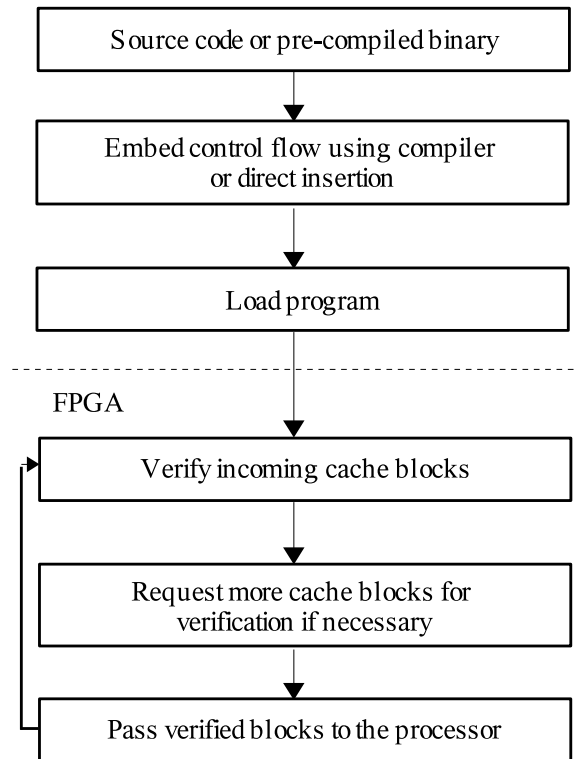
```
┌─────────────────────────────────────────┐
│    Source code or pre-compiled binary    │
└─────────────────────────────────────────┘
                    │
                    ▼
┌─────────────────────────────────────────┐
│    Embed control flow using compiler     │
│           or direct insertion            │
└─────────────────────────────────────────┘
                    │
                    ▼
┌─────────────────────────────────────────┐
│              Load program                │
└─────────────────────────────────────────┘
 - - - - - - - - - - - │ - - - - - - - - - -
   FPGA                │
                       ▼
┌─────────────────────────────────────────┐
│       Verify incoming cache blocks       │◄─┐
└─────────────────────────────────────────┘  │
                    │                         │
                    ▼                         │
┌─────────────────────────────────────────┐  │
│        Request more cache blocks for     │  │
│           verification if necessary      │  │
└─────────────────────────────────────────┘  │
                    │                         │
                    ▼                         │
┌─────────────────────────────────────────┐  │
│     Pass verified blocks to the processor│──┘
└─────────────────────────────────────────┘
```

Figure 2: CODESSEAL: System View

and data are visible "below" the FPGA, typically inside a chip, thereby preventing an attack that sniffs the address/data lines between processor and memory. The original code and data are encrypted by a compiler that uses the same keys. The assumption is that both FPGA loading and compilation occur at a safe site prior to system operation.

Note that FPGA's can be used to detect code tampering even when full encryption is not used. A simple approach, in this case, is to compute hashes of instruction code blocks. For example, instructions block hashes using SHA-1 are maintained inside the FPGA, and as each instruction block is loaded, the SHA-1 hash is computed in the FPGA. If the hash does not match the stored hash, tampering is assumed and the processor is halted. Even when full encryption is used, it is desirable to perform a hash check because a tampering attack can be used to disrupt execution without decryption. Our experiments have revealed that this technique results in negligible performance overhead for most of the Spec and Media benchmarks. While this technique maintains code integrity it does not prevent structural (control flow) attacks. We are currently developing and testing techniques to prevent structural attacks by embedding the control flow information into the code.

Data tampering, in encrypted systems, is more complicated because a write operation necessitates a change in the encryption: data needs to be re-encrypted on write-back to RAM. Also, because data can get significantly larger than code, a large set of keys might be needed to encrypt data, resulting in a key management problem. Our approach to this problem uses the concept of a "key page table".

8

4 Future Research Directions

In this section, we briefly identify future research directions in the area we have termed security-driven compilation:

- *Integrated approaches*. Clearly, it is desirable to combine the simple efficiency of obfuscation and checksum methods with hardware approaches. Integrated approaches also have gthe advantage of creating executables that flexibly execute on both on processors with supporting security hardware as well as those without.

- *Security-performance tradeoffs*. While the performance impact of individual techniques have been studied [8, 29, 35, 65], future compilers should provide system designers with features that allow explicit tradeoff between security and performance that go beyond key length. For example, a combination of partial encryption and checksum-checking may suffice in providing a wide range of execution performance. Indeed, very little is known today about the relative interplay betwween standard compiler optimizations and security-driven code transformations.

- *Attacks on encrypted execution*. Just because executables are encrypted in main memory during execution doesn't mean attacks cannot be effective. Several types of replay, data and structural attacks are possible that a resourceful attacker can use to systematically uncover particular behaviors. We term such attacks as EED attacks – attacks on *Encrypted Executables and Data*. To help detect such attacks, compilers will need to play a key role in extracting structural information for use by supporting hardware.

- *Joint Compiler/OS approaches*. A secure or trusted operating system provides an infrastructure that may be more flexible than hardware and have broader impact so that files and storage may be included in security guarantees. Thus, for example, compiler-generated keys can be used in file read and write operations to secure files.

- *Secure tool-chains*. As mentioned in the Introduction, compilers have the potential of incorporating security in large numbers of systems because of their unique role in the creation of software. Similarly, one can conceive of incorporating security as a major design objective in the entire tool chain for computing infrastructures. A deeper issue arises from asking the question: how can a tool-chain itself provide guarantees? Trusted tool-chains may form the first line of defense prior to system operation.

- *Hardware/Software co-design*. Just as hardware/software co-design is today becoming the prevalent approach to designing systems, so should this approach prove valuable in designing trusted systems. The design of supporting hardware will depend on the software application, and likewise, compiler optimizations will depend on supporting hardware.

5 Summary

This paper has surveyed several projects in the rich and growing intersection between compilers and security. We have argued that compilers are especially well-suited to addressing many problems in software protection and that compiler research in this area can extend in many promising new directions.

References

[1] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the Spi calculus. *Proc. of the Fourth ACM Conference on Computer and Communications Security,* ACM Press, 36–47, 1997.

[2] A.W. Appel and E.W. Felten. Proof-Carrying Authentication., *6th ACM Conference on Computer and Communications Security*, November 1999.

[3] W. Arbaugh. A Secure and Reliable Bootstrap Architecture, *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997, pp 65–71.

[4] D. Aucsmith. Tamper resistant software: An implementation. in Anderson,R., Ed., Information Hiding, First International Workshop, Cambridge, UK, 1996, Springer-Verlag Lecture Notes in Computer Science, Vol. 1174, pp. 317–333.

[5] D. Balfanz, D. Dean, M. Spreitzer. A Security Infrastructure for Distributed Java Applications. *Proceedings of 2000 IEEE Symposium on Security and Privacy*, May, 2000.

[6] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *Proc. CRYPTO 2001*, August 2001.

[7] L. Bauer, M. Schneider and E.W. Felten. A Proof-Carrying Authorization System. Technical report CS-TR-638-01, Department of Computer Science, Princeton University, April 2001.

[8] H.Chang and M.J.Atallah. Protecting software code by guards. *ACM Workshop on Security and Privacy in Digital Rights Management*, Philadelphia, 2001.

[9] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. *ACM CCS, 2002.*

[10] S.Cheng, P.Litva and A.Main. Trusting DRM software. *Workshop on Digital Rights Management for the Web*, January 2001, France.

[11] C.Collberg, C.Thomborson and D.Low. Breaking abstractions and unstructuring data structures. *Proc. IEEE International Conference on Computer Languages, ICCL'98*, Chicago, IL, May 1998.

[12] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[13] C. Collberg, and C. Thomborson. Software watermarking: Models and Dynamic Embeddings. *Proc. 26th ACM SIGPLAN-SIGACT on principles of Programming languages (POPL'99)*, 311–324, 1999.

[14] C. Collberg, and C. Thomborson. Watermarking, Tamper-proofing, Obfuscation: Tools for Software Protection. Technical report 2000-03, University of Arizona, 2000.

[15] CoreDES data sheet v2.0. Actel Corporation, 2003. www.actel.com

[16] C.Cowan. Software Security for Open-Source Systems. *IEEE Security and Privacy*, Jan/Feb 2003, pp. 38-43.

[17] C. Cowan. FormatGuard: Automatic protection from printf format string vulnerabilities. *Proc. USENIX Security Symposium,* Washington, DC, August 2001.

[18] J. Daemen and V. Rijmen, The Block Cipher Rijndael, Smart Card Research and Applications, LNCS 1820, J.-J. Quisquater and B. Schneier, Eds., Springer-Verlag, 2000, pp. 288-296.

[19] A. Dandalis, V. K. Prasanna, and J. D. P. Rolim. An adaptive cryptograhpic engine for IPSec Architectures. In *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2000.

[20] Design security with Actel FPGAs. Actel Corporation, 2003. www.actel.com

[21] J. Dyer, M. Lindemann, R. Perez, R. Sailer, S.W. Smith, L. van Doorn, S. Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer. 34: 57-66.* October 2001.

[22] J. Dyer, R. Perez, S.W. Smith, M. Lindemann. "Application Support Architecture for a High-Performance, Programmable Secure Coprocessor." *22nd National Information Systems Security Conference.* October 1999.

[23] U. Erlingson and F.B. Schneider. IRM enforcement of java stack inspection. *IEEE Symposium on Security and Privacy,* Oakland, California, May 2000.

[24] U. Erlingson and F.B. Schneider. SASI enforcement of security policies: A retrospective. *Proceedings of the New Security Paradigms Workshop,* Ontario, Canada, Sept 1999.

[25] M. Fisher. Protecting binary executables. *Embedded Systems Programming*, Vol. 13, No. 2, February 2000.

[26] J. S. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* Atlanta, Georgia, 1999.

[27] A. Gordon and D. Smye. Typing a multilanguage intermediate code. *Conference Record of POPL 2001: 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* 248–260, 2001.

[28] D. Grossman and G. Morrisett. Scalable certification for typed assembly language. *2000 ACM SIGPLAN Workshop on Types in Compilation,* Montreal, Canada, 2000.

[29] B. Horne, L. R. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. *ACM Workshop on Security and Privacy in Digital Rights Management*, November 2001.

[30] http://www.research.ibm.com/secure_systems.

[31] T.Jim, G.Morrisett, D.Grossman, M.Micks, J.Cheney and Y.Wang. Cyclone: a safe dialect of C. *Usenix*, June 2002, Monterrey, CA, pp.275-288.

[32] A. Jones, D. Bagchi, S. Pal, X. Tang, A. Choudhary, and P. Banerjee. PACT HDL: a C compiler with power and performance optimizations. In *Proc. of the Int'l Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, October 2002.

[33] J. Kaps and C. Paar. Fast DES implementations for FPGAs and its application to a universal key-search machine. *Selected Areas in Cryptography*, 1998.

[34] D.Kozen. Language-based security. *Proc. Conf. Math. Foundations of Computer Science*, September 1999, pp. 284-298.

[35] D. Lie, C. Thekkath, M. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proc. of the 9th Int'l Conference Architectural Support for Programming Languages and Operating Systems (ASPLOX-IX)*, November 2000.

[36] P.A.Loscocco and S.D.Smalley. Meeting critical security objectives with security-enhanced Linux. *Proc. Linux Symposium*, Ottawa, 2001.

[37] G.Morrisett, D.Walker, K.Crary and N.Glew. From system F to typed assembly language. *ACM Trans. Prog. Lang.*, Vol. 21, No. 3, pp.528-569, May 1999.

[38] G. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking, *OSDI'96*.

[39] G.Necula, S.McPeak and W.Weimer. CCured: type-safe retrofitting of legacy code. *Principles of Programming Languages*, 2002, pp. 128-139.

[40] G. Necula. Proof-Carrying Code, *Proceedings of POPL'97*.

[41] G. Necula. http://www.cs.berkeley.edu/ nekula

[42] P.Ott, A.Choudhary, B.Narahari, R.Simha and J.Zambreno. Compiler/Hardware Solutions to Attacks on Encrypted Executables. In preparation.

[43] E.R. Palmer. An Introduction to Citadel—A Secure Crypto Coprocessor for Workstations. Research Report RC 18373, IBM T.J. Watson Research Center, 1992.

[44] V. K. Prasanna and A. Dandalis. FPGA-based cryptography for internet security. *Online Symposium for Electronic Engineers*, November 2000.

[45] F.B.Schneider, G.Morrisett and R.Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead*, Lecture Notes in Computer Science, Vol. 2000, Springer-Verlag, pp. 86-101.

[46] U.Shankar, K. Talwar, J.S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. *10th Usenix Security Symposium,* 2001.

[47] S. Smith and V. Austel. Trusting Trusted Hardware: Towards a Formal Model of Programmable Secure Coprocessors, *Proc. 3rd Usenix Workshop on Electronic Commerce*, August 1998.

[48] S. Smith, R. Perez, W. Weingart and V. Austel, Validating a High Performance, Programmable Secure Coprocessor, IBM Research Report, RC 21416, 15 February 1999.

[49] S. Smith and S. Weingart, Building a High-Performache Programmable Secure Coprocessor, *Computer Networks*, Vol. 31, pp 831–860, 1999.

[50] S. Smith. Secure coprocessing applications and research issues. Los Alamos Unclassified Release LA-UR-96-2805, 1996.

[51] D. Stinon. *Cryptography: Theory and Practice*. Chapman and Hall, 2002.

[52] R. Taylor and S. Goldstein. A high-performance flexible architecture for cryptography. In *Proc. of the Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, August 1999.

[53] J.D. Tygar and B.S. Yee. Dyad: A System for Using Physically Secure Coprocessors. Harvard-MIT Workshop on Protection of Intellectual Property. April 1993.

[54] Tygar, J.D., Yee, B., Dyad: A System for Using Physically Secure Copropcessors, CMU-CS-91-140R, Carnegie Mellon University, Pittsburgh, PA. 1991.

[55] P. Wagle, C. Cowan. Stackguard: Simple stack smash protection for GCC. *Proc. of the GCC Developers Summit,* 243–256, 2003.

[56] D. Wagner, J. Foster, E.A. Brewer, A. Aiken. A first step towards autoamted detection of buffer overrun vulnerabilities. *Proc. Network and Distributed Systems Security,NDSS 2000.*

[57] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing the static analysis of programs. Technical Report, Dept of Computer Science, CS-2000-12, University of Virginia, 2000.

[58] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. *Proc. of 2001 IEEE/IFIP International conference on dependable systems and Networks (DSN'01)*, Sweden, 2001.

[59] Weingart, S., Physical Security for the mABYSS System, *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, 1987, pp 52-58.

[60] S.H. Weingart, S.R. White, W.C. Arnold, G.P. Double. An Evaluation System for the Physical Security of Computing Systems. *6th Computer Security Applications Conference*. 1990.

[61] White, S., Comfortd, L, Abyss: A Trusted Architecture for Software Protection, *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, 1987, pp 38-51.

[62] J. Wyant. Establishing security requirements for more effective and scalable DRM solutions. *Workshop on Digital Rights Management for the Web*, January 2001.

[63] Yee, B., and Tygar, J.D., Secure Coprocessors in Electronic Commerce Applications, *Proc. First USENIX Workshop on Electronic Commerce*, July 1995, 155-170.

[64] B.S. Yee. Using Secure Coprocessors. Ph.D. Thesis, Computer Science Technical Report, CMU-CS-94-149. Carnegie Mellon University,1994.

[65] J.Zambreno, A.Choudhary, R.Simha, B.Narahari and N.Memon. SAFE-OPS: A Compiler/Architecture Approach to Embedded Software Security. *ACM Trans. Embedded Computing*, accepted.

[66] J.Zambreno, A.Choudhary, B.Narahari and R.Simha. Flexible Software Protection Using Hardware/Software Codesign Techniques. In *Proceedings of Design, Automation and Test in Europe (DATE)*, 2004

[67] X.Zhuang, T.Zhang, H-H.Lee and S.Pande. Hardware Assisted Control Flow Obfuscation for Embedded Processors. *CASES 2004*, Washington DC, Sept. 2004

[68] X.Zhuang, T.Zhang and S.Pande. HIDE: An Infrastructure for Effi ciently Protecting Information Leakage on the Address Bus. *ASPLOS 2004*, Boston, MA., Oct 2004.

[69] Trusted Computing Platform Alliance. `http://www.trustedcomputing.org`.

[70] Cyberinfrastructure Report, National Science Foundation, 2003.