Compiler-Directed Region-Based Security for Low-Overhead Software Protection

Vijay Kongubangaram
Department of Computer Science
The George Washington University
Washington, DC
vijaykpk@gwu.edu

Olga Gelbart
Department of Computer Science
The George Washington University
Washington, DC
rosa@gwu.edu

Rahul Simha
Department of Computer Science
The George Washington University
Washington, DC
simha@gwu.edu

Bhagirath Narahari
Department of Computer Science
The George Washington University
Washington, DC
narahari@gwu.edu

Abstract

Software security has become a prominent area of research in recent years, with research efforts spanning a wide range of topics. Among these are techniques such as those in this paper that are in the general area of languages, compilers and architecture aimed at increasing the security of computing systems. This paper describes a compiler technique that performs risk-analysis on source code and generates an encrypted executable that both provides security but yet reduces overhead by selectively encrypting low-risk portions with less overhead. Regions of the code that are more vulnerable receive a higher degree of encryption. Experimental results for this technique, which we call Region-Based Security, using a collection of benchmarks show that execution overhead is reduced considerably by using this approach.

1 Introduction

Attackers exploit software vulnerabilities caused by programming errors, and system or programming language flaws. Since the worst of these exploits occur during the operation of the system, it is crucial to verify the integrity of executing software at the time of execution. Sophisticated attackers are able to tamper with hardware in order to alter execution during runtime. Many software and combined software-hardware approaches have been proposed to detect these attacks [4, 1, 3, 5]. CODESSEAL is one such tool. It is an approach that combines static and dynamic verification methods with compiler techniques and a processor supplemented with a secure hardware component in the form of an FPGA (Field Programmable Gate Array). This combination of compiler-instrumented executables and accompanying hardware-support has been shown to provide a secure execution environment for fully encrypted execution [11]. The tool incorporates techniques to prevent code tampering, code understanding, and several types of replay, data and structural attacks.

Several code security approaches like tamper resistant packaging, copyright notices, guards, code obfuscation, register encoding [4, 1, 3, 8, 12], focus on providing solutions for a specific type of vulnerability and are susceptible to code tampering and code injection by sophisticated attackers. These approaches suggest applying the techniques to the entire system. As the entire system may not be vulnerable to attacks, it is possible to extract efficiency by concentrating high-overhead encryption on those components more vulnerable than others. We instrument a compiler with Region-Based Security (RBS), in which basic blocks are assessed for their risk level and correspondingly secure using different mechanisms. We integrate RBS into the CODESSEAL suite of tools to provide an experimental platform in which to evaluate this approach. With this mechanism, CODESSEAL breaks the code into regions of different vulnerabilities and applies different techniques such as instruction hashing, instruction and data encryption, or control flow protection The results presented in this paper show considerable decrease in overhead over applying security mechanisms equally on the entire system.

Finally, we also point out that our approach brings to-

gether two diverse strands of research in the software protection area. The first strand consists of static risk-analyzing tools such as FlawFinder, MOPS or ITS4[2, 6, 4, 13, 14]. The second consists of a variety of compiler and compiler-hardware approaches, such as [3, 1, 5, 11] that instrument code with checksums or encrypts code for fully-encrypted execution. By using the results of a static analyzer and refining the results to apply to the basic blocks of executable code generated by a compiler, a compiler-hardware software protection mechanism can carefully target the application of security mechanism to help manage the tradeoffs between security and performance.

This paper is organized to present the CODESSEAL architecture in Section 2, which is the framework on which RBS is implemented. The presentation is self-contained. The key ideas and implementation details are presented in Section 3. Section 4 gives the results and Section 5 concludes the paper with details of future work.

2 CODESSEAL

CODESSEAL (COmpiler DEvelopment Suite for SECure AppLications) is an infrastructure focused on joint compiler/hardware techniques for fully encrypted execution, in which the program and data are always in encrypted form in memory. Encrypted execution is preferred for highly-secure applications in which guarantees against both disruption and loss of intellectual property are desired. However, as is well-known now, simply encrypted execution (keeping instructions and data in encrypted form) alone does not prevent all forms of attack. Several types of replay, data and structural attacks, such as control-flow attacks, are known. These attacks have been termed Encrypted Executable and Data (EED) attacks [11]. EED attacks exploit structure vulnerabilities in encrypted instruction streams and data that can be uncovered by direct manipulation of hardware (such as address bus manipulation) in a well-equipped laboratory. To help detect such attacks, the CODESSEAL approach makes use of a combination of compiler-directed encryption and supporting hardware that maintains and checks structural information as well as data integrity.

2.1 Architecture

The CODESSEAL framework has two main components: (1) static verification and (2) dynamic verification. Static integrity and control flow information are embedded into the executable during compilation. The security module is responsible for applying the security techniques such as encryption, hashing. The static verification module checks the overall integrity of the executable and signature [10]. Upon success, the executable is launched and each block is dynamically verified in the supporting hardware, in this case reconfigurable logic (FPGA) that is itself programmed to provide this support. The dynamic verification module is responsible for preventing run-time attacks on the program. The dynamic verification module has two functions: (1) check that code and data blocks have not been modified at run-time by an attacker and (2) assert legal control flow in the program. Any changes made to the control flow graph of the program is considered equivalent to code tampering, following which the program is halted.

The CODESSEAL hardware architecture is shown in the Figure 1. One advantage of using an FPGA is that the security mechanisms and cryptographic algorithms can be reprogrammed as they change, or even customized to each application. The FPGA is placed between the main memory and the cache that is closest to the main memory (either L1 or L2, depending on the system)(Figure 2). The instructions and data are loaded into the FPGA in blocks. Decryption and other security-related checks such as control flow verification are performed in the FPGA. Thus, the decrypted code and data are visible only inside the chip, thereby defeating an attacker who sniffs the address/data lines between processor and memory.

2.2 Security Techniques

CODESSEAL provides several security mechanisms to protect against EED attacks. These mechanisms include instruction and data encryption, instruction hashing, control flow protection using hardware stack, all of which protect the system from the sophisticated attackers who have access to the hardware of the system. CODESSEAL starts by fully encrypting each executable at compile time with the assumption that decryption will be performed by the FPGA at runtime. CODESSEAL also assumes that keys are loaded into the FPGA securely, either once in a secure location or at runtime using a secure load of the FPGA's configuration. The compiler also generates a hash for each code block. As pointed out in [11], the hash maintains code integrity and encryption protects against loss of intellectual property. Instruction and data block hashes (using SHA-1, for example) are maintained inside the FPGA or with the basic blocks and verified each time a new block is loaded. If the computed hash does not match the stored hash, the processor is halted. However, neither technique prevents structural (control flow) attacks.

The control-flow verification is provided using hardware stack. A replica of hardware stack is implemented in the FPGA which stores the return address of each function call. On each function return, the return address is compared with the top of the stack. This assures that the function is returning to the correct position in the control flow. By using additional hardware to verify the program at runtime,

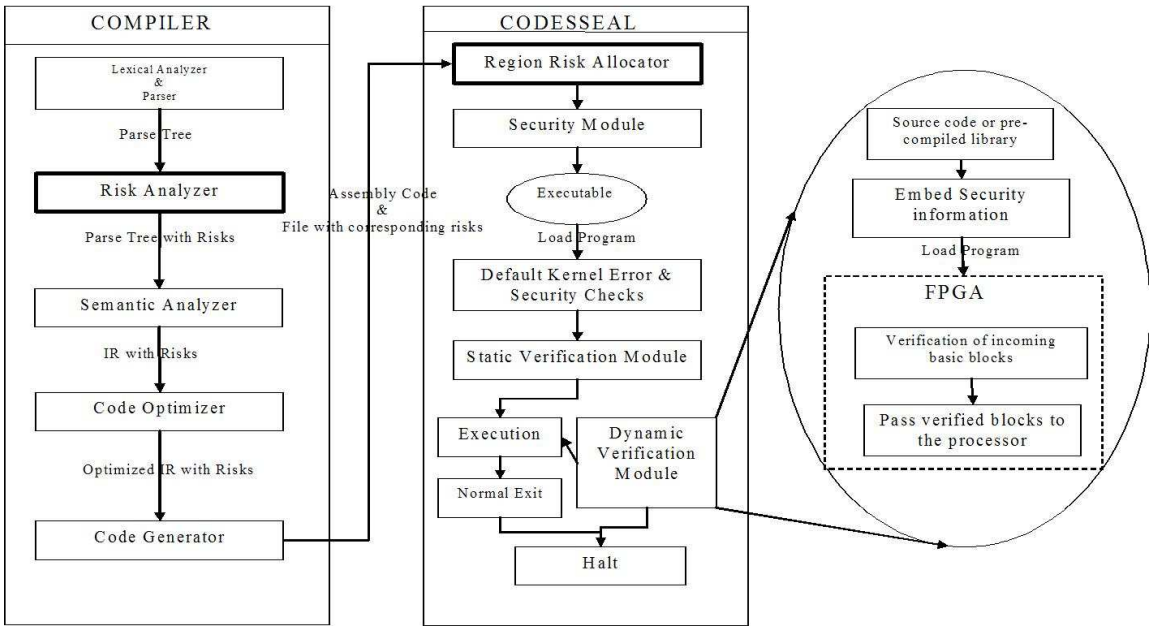

Figure 1. CODESSEAL with Region-Based Security

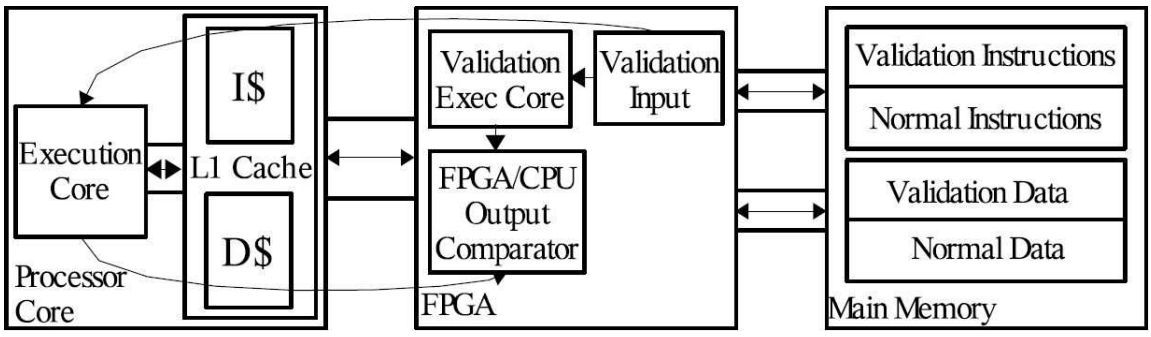

Figure 2. Hardware design - FPGA in between Processor Core and Main Memory

adding additional code to the executable can be avoided and thus preventing code analysis attacks.

Encryption is also performed on the data to protect the system from attacks such as buffer overflow and data manipulation. In this mechanism, the data stored in the memory is always encrypted and signed. This makes it harder for the attacker to change the data to alter the execution. Whenever data is written back to the memory, the FPGA encrypts and sends back the data to the memory. When a data block is read from the memory, the FPGA decrypts and compares the hash of the block. If the block doesn't match its hash, the execution will be halted.

Thus, CODESSEAL provides data and instruction encryption, instruction hashing and control flow protection security mechanisms which focus on protecting different kinds of vulnerabilities. However, these techniques carry a significant overhead. Some of the overhead can be mitigated using hardware techniques such as pipelining and by using a large cache in the FPGA. Even so, there is still some overhead incurred when blocks are decrypted for the first time, or whenever blocks are re-loaded. The purpose of this paper is to explore a software approach to lower this overhead, using compiler-directed region-based security.

3 Region Based Security

We propose Region-Based Security, a compiler-driven approach that combines risk-analysis and selective protection to help reduce the overhead in encrypted execution platforms. The key idea is that each block of code may be assessed for its vulnerability, following which protection is applied selectively. For example, variable declarations and mathematical operations are reportedly not as susceptible as control or data instructions. Hence, it makes sense to accord these more susceptible instructions a higher degree of protection. We contrast this new approach with the standard encrypted-execution approach of protecting the entire executable.

3.1 Risk Analysis

We add a risk analysis module to the compiler. This module is responsible for assessing the risk inherent in each block of code. Note that our paper does not focus on novel risk analysis techniques; we assume that any risk-analysis module can be used. Several static risk analyzers are in use today, some of which have been commercially successful. After analyzing the vulnerabilities, the module assigns different risks to different instructions. It maintains a list of known vulnerable functions and based on the risks presented by the functions, each is associated with a different risk level.

The proposed RBS mechanism uses four different risk levels: 'Low', 'High', 'Undecided' and 'Neutral'. In this scheme, risk levels 'High' and 'Low' are assigned to instructions based on how static-analzyers rate the functions these instructions were compiled from. The 'Neutral' risk level is assigned to instructions such as declarations and mathematical operations which do not pose much of a threat based on known attacks. The risk level 'Undecided' is assigned to any instruction for which risk level cannot be determined by the risk analysis alogrithm.

As mentioned, static analysis of source code is performed to find the vulnerabilities. The Risk Analyzer module identifies vulnerabilities, for example, in library functions such as strcpy(), memcpy(), or sprintf(). This module is inserted after the parser module of the compiler (Figure 1). During the analysis of the parse tree, if the module finds a tree node which uses one of these known vulnerable functions, the module finds the risk level associated with this function and assigns the tree node with the risk level. This modified parse tree with the risk levels is passed on to the semantic analyzer. The semantic analyzer module generates an Intermediate Representation (IR) with the associated risks. This IR is sent to the code optimizer module which generates the optimized IR with corresponding risk levels. Then, the assembler takes in the optimized IR with the risks and produces a file with the risks associated with each assembly instruction. This file is passed to the CODESSEAL framework for assigning regions with risks. Note that all the instructions in a function are assigned a risk level no higher than the risk level assigned to the function. We point out that several other types of vulnerable patterns are presented in [9, 7, 13, 14, 2, 6]. These ideas can easily be incorporated in our approach as alternative types of risk analysis.

3.2 Region Risk Allocator

The Region Risk Allocator module is responsible for breaking the code into different regions and assigning risks to these regions based on the instruction-level risk values identified earlier. This module is placed in the CODESSEAL framework (Figure 1) at the beginning of the tool chain, so that the regions can be assigned with the risks. Then the framework can decide which security technique is best suited to protect against the risk presented by the region. A mapping between risk level and the security mechanism is made, enabling the framework to choose the techniques based on the risk levels. The security module applies these techniques and embeds the risk level in to the region so that the static and the dynamic verification modules can check the validity of the region based on the risk level embedded there.

The regions in RBS are the basic blocks generated by the backend of the compiler. The region risk allocator conser-

vatively assigns each basic block with the *highest* risk level of the instructions in that block. After each basic block has a risk level assigned, the security module applies the security mechanism corresponding to the risk level. It also embeds the risk level in the basic block so that the static and dynamic verification modules can extract the risk level and validate the basic block.

4 Results

4.1 Experimental Setup

We now describe our experimental setup. We used the SimpleScalar 3.0 architecture simulator configured for the ARM Processor (ARM1020E core, 400 MHz). The `gcc` V3.3 ARM cross compiler was used for static compilation of the benchmarks. The FPGA chosen was modeled after the Virtex-II XC2V800 (200 MHz, 3 MB memory). 32 byte caches were used that run write-through and LRU Replacement policies. The main memory parameters were: 100 Mhz, with 24 processor cycles delay for first time access and 4 cycles for subsequent accesses. Branch prediction in SimpleScalar was turned off and the FPGA was called on every instruction or data cache miss.

Encryption was performed using an implementation of the AES algorithm that operates on 128-bit blocks with 40 processor cycles delay per block. We used an implementation of SHA-1 hashing that takes 164 processor cycles for the hash calculation and two cycles for hash comparison. The hashes are stored in each block or in the FPGA. The FPGA performs hash verification as each block loads. The risk levels are stored in the basic block and the mapping table is stored in the FPGA. Following an L1 cache miss, when a block is brought in, the block's hash verification is performed. This involves three steps: (1) the hash of the block is calculated, (2) the corresponding block's hash is fetched from either the FPGA memory or from basic block itself, and (3) the two hashes are compared.

The experiments used a database of functions and their risk levels. The table 2 gives the risk level and the corresponding security mechanism used. All the three approaches assign Instruction Encryption and Hashing for 'Low' risk basic blocks; and Instruction Encyrption, Hashing and Control Flow protection using hardware stack for the basic blocks with risk level 'High'. 'Neutral' basic blocks in the three approaches are not assigned any security mechanism because they are considered harmless. The risk level 'Undecided' in the first approach is conservatively assigned all the security mechanism: Instruction and Data Encryption, Instruction and Data Hashing and Control Flow Protection etc. In the second and third approaches, the 'Undecided' basic blocks are assigned the same security mechanism as 'Low' and 'High' risk level respectively.

The benchmark suites used in the experiment are DIS and Media. All the benchmarks are run through the CODESSEAL framework with Region Based Security enabled. The table 1 gives the number of regions (basic blocks) executed for each different risk level. The table shows the penalties when running different schemes. The figure 3 presents the effectiveness of RBS in decreasing the number of basic blocks that need security. The comparisons are made against a baseline with no protection. A discussion of these results is presented in the next section.

4.2 Analysis

The results presented in the paper evaluate the efficiency of the Region Based Scheme on two metrics: the number of basic blocks that need protection and the execution penalty (overhead). A decrease in the number of blocks needing protection will decrease the execution time, power consumption and memory usage of the embedded system. The figure 3 gives the decrease in number of basic blocks that are vulnerable. The RBS mechanism decreases vulnerable blocks by 11% to 24 %. The decrease is due to the recognition of some basic blocks as no-threat regions i.e. these regions do not pose any threat and hence can be safely assumed to require no protection.

The table 1 shows the efficiency of the RBS scheme in identifying the different threat regions. Based on the vulnerability of functions and their risk levels, every instruction in the program is assigned a risk level. When the program is divided into basic blocks, the maximum risk of the instruction in the region becomes the region's risk level. In all the benchmarks, a considerable number of basic blocks have risk level 'Neutral'. These pose no threat. There is also a considerable number of basic blocks with risk level 'High'. Most of the functions the benchmarks use, such as file operations and IO operations, are all assigned the risk level 'High' and hence the large number. The 'Undecided' regions also share a major part of the basic blocks. Since the system cannot assess the vulnerabilities of these regions, three different approaches have been chosen. As the risk level is uncertain, one approach conservatively chooses all the security mechanisms in order to protect against all attacks. Second approach chooses the same security mechanism as the 'Low' risk level in an assumption that 'Undecided' risk level may not be need more protection than the low risk level basic blocks. This assumption may be unrealistic and hence a third approach is designed that assigns the same security mechanism as the 'High' risk level assuming that 'Undecided' blocks are as vulnerable as the 'High' risk level basic blocks. The table 2 presents a sample mapping between risk levels and security mechanisms. Approach I assign all the schemes to 'Undecided' risk level, Approach II employs the 'Low' security mechanism and Approach III

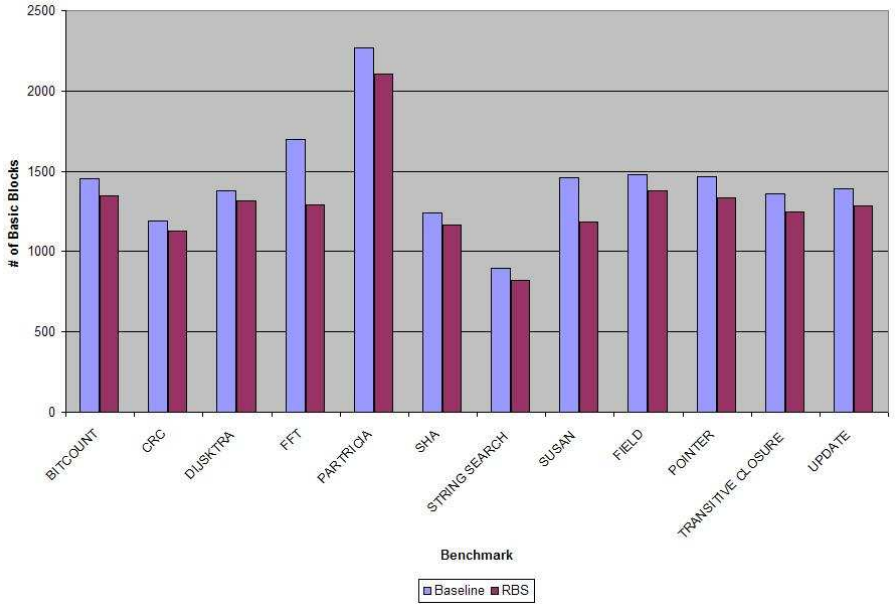

Figure 3. Decrease in Basic Blocks requiring security

	Unique Basic	Number of Basic Blocks with Risks			
Benchmark	**Blocks Executed**	**Undecided**	**Neutral**	**Low**	**High**
Bitcount	1451	431	105	0	915
Crc	1190	485	65	2	638
Dijkstra	1381	467	68	2	844
FFT	1700	454	411	1	835
Patricia	2270	454	164	5	1647
SHA	1238	517	71	4	646
String Search	899	410	78	0	411
Susan	1464	448	277	15	723
Field	1481	480	104	0	897
Pointer	1464	497	129	0	838
Transitive Closure	1359	487	109	0	763
Update	1389	483	106	0	800

Table 1. Number of Unique Basic Blocks executed for each risk level

	Security Mechanism		
Risk Level	**Approach I**	**Approach II**	**Approach III**
Undecided	All Schemes	Encryption and Hashing	Encryption and Hashing Control Flow Protection
Neutral	-	-	-
Low	Encryption and Hashing	Encryption and Hashing	Encryption and Hashing
High	Encryption and Hashing, Control Flow Protection	Encryption and Hashing, Control Flow Protection	Encryption and Hashing, Control Flow Protection

Table 2. Mapping between risk levels and security mechanisms

Benchmark	Entire System	Approach I	Approach II	Approach III
Bitcount	1.62	1.50	1.02	1.49
Crc	12.00	11.34	6.45	11.34
Dijkstra	17.59	6.52	0.71	0.88
FFT	24.73	8.84	2.35	3.46
Patricia	13.86	6.38	3.67	4.59
SHA	0.31	0.27	0.17	0.24
String Search	10.78	7.25	3.89	4.66
Susan	4.79	2.09	0.97	0.99
Field	1.08	0.61	0.27	0.41
Pointer	24.3	9.75	1.57	2.39
Transitive Closure	432.12	154.94	0.12	0.14
Update	31.65	14.33	3.82	5.33

Table 3. Execution Penalties of three approaches over baseline.

employs the 'High' security mechanism for the 'Undecided' risk level. These 'Undecided' instructions are generated by the loader module of the compiler, on which the system does not have control. In the future, the loader will be modified to assign risk levels to any instruction that it generates. This would decrease the number of regions with risk level 'Undecided' and could further decrease the overhead.

The table 3 presents the results of the three approaches presented in the table 2. In the 'Entire System' approach, the entire system is protected by all schemes implemented in CODESSEAL. The penalities are calculated over the baseline execution without any security protection. Applying protection to entire system has the highest overhead. The results indicate that applying all the schemes for the 'Undecided' risk level (Approach I) has a high overhead over the other two approaches. Approach I employs all the security mechanism for the 'Undecided' risk level and this increases the overhead. An unusual penalty increase can be seen in the transtive closure benchmark. The reason is due to the application of data encryption and data hashing mechanism on the large amount of data used by this benchmark. This increase is not seen in the other approaches as they don't use data security mechanisms. Hence, the overhead can be controlled by changing the security mechanisms. Approach II has lower overhead than Approach III, as the security mechanism applied for 'Undecided' risk level in Approach II produces lower overhead but also provides lower protection. A considerable amount of decrease in overhead can be seen when RBS scheme is adopted in the three approaches. The RBS schemes decreases the overhead considerably when compared to applying all the schemes and at the same time providing the same level of security. Different approaches produce different overheads; this gives the desingner a choice - if one wants to decrease the overhead, one can weaken the security and vice versa.

The results presented in the paper show that RBS is effective as an approach, and because it is complementary to the actual security mechanisms, can be implemented as an independent compiler module. An RBS optimization also leads to lower power consumption and memory usage, both valuable resources in embedded systems.

5 Conclusion and Future Work

Region Based Security (RBS) is a compiler-level tool for encrypted execution plaforms that allows careful tradeoff of security and performance. The approach uses the output of well-known risk-assessing static analysis methods to selectively apply a suite of security techniques that match risk-level with the strength of the security technique. We incorporated RBS into the CODESSEAL framework and used benchmarks to evaluate the effectiveness of RBS. Experimental results show a considerable decrease in overhead. Future research work will focus on incorporating a wider variety of risk analysis, incorporating assessments from multiple static analysis tools.

Acknowledgements: This work is partially supported by NSF Grant ITR-0325207 and AFOSR grant FA9550-06-1-0152.

References

[1] H. Chang and M. J. Atallah. Protecting software code by guards. In *DRM '01: Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, pages 160–175, London, UK, 2002. Springer-Verlag.

[2] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *CCS '02: Proceedings of the 9th ACM conference on Computer and com-*

munications security, pages 235–244, New York, NY, USA, 2002. ACM Press.

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

[4] C. Cowan. Software security for open-source systems, 2003. IEEE J. Security and Privacy, 1(1):38–45,.

[5] C. T. David Lie, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 168–177, New York, NY, USA, 2000. ACM Press.

[6] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, San Diego, CA*, October 2000.

[7] D. Evans. Splint. http://www.splint.org.

[8] M. Fisher. Protecting binary executables. *Embedded Systems Programming.*, 13(2), 2000.

[9] J. Foster. *Type qualifiers: Lightweight specifications to improve software quality*. PhD thesis, University of California, Berkeley, 2002.

[10] O. Gelbart, B. Narahari, and R. Simha. Spee: A secure program execution environment tool using static and dynamic code verification. In *Proc. the 3rd Trusted Internet Workshop. International High Performance Computing Conference. Bangalore, India*, 2004.

[11] O. Gelbart, P. Ott, B. Narahari, R. Simha, A. N. Choudhary, and J. Zambreno. Codesseal: Compiler/fpga approach to secure applications. In *ISI*, pages 530–535, 2005.

[12] K. Mohan, B. Narahari, R. Simha, P. Ott, A. N. Choudhary, and J. Zambreno. Performance study of a compiler/hardware approach to embedded systems security. In *ISI*, pages 543–548, 2005.

[13] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. *ACM Transactions on Information and System Security*, 5(2), 2002.

[14] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.