Poster: Experiments in Hardware Security Tagging

J. Song (Ph.D. student) and S. Zakeri (M.S. student), J. Alves-Foss (faculty),
Center for Secure and Dependable Systems,
University of Idaho
Moscow, ID USA
song3202@vandals.uidaho.edu, zake7991@vandals.uidaho.edu, jimaf@uidaho.edu

***Abstract*—To enhance the security of computer systems, and to take some of the burden off of the software developers, researchers are looking at hardware-based security tagging schemes to enhance system security. The research highlighted in this poster addresses the evaluation, design and implementation of tagging schemes for access control and information flow; specifically the implementation at the assembly language level for a zero-kernel operating system. We highlight key lessons learned that we have not seen addressed in related literature.**

Keywords—hardware tagging, RTEMS, SPARC, zero-kernel operating system (ZKOS)

I. INTRODUCTION

Researchers and hardware developers have been exploring techniques to enhance the security of our computer systems. They are painfully aware that even the best software developers make mistakes, and with an internet user base of billions of inter-connected people, there are just too many opportunities for our systems to come under attack. One of the technologies being explored involves the use of a data tagging scheme. These schemes attach security labels to memory regions and processor registers to carry information about tagged data during program execution. They can be used to ensure the semantics of computations are correctly implemented; to isolate code and data, users and system; or be used to enforce security policies at the hardware level. The implementation of tagging in hardware provides developers with enhanced security mechanisms with improved performance, as compared to traditional microprocessors.

We have been investigating these tagging schemes through a multi-part research project. The first part involves the modification of a hardware simulator, based on the Open SPARC microprocessors and the SIS simulator, to provide a template for the insertion of different tagging engines. The second part involves the development of our own tagging scheme. Our initial implementation was based on 'C' programming language features and then it was mapped down to the SPARC assembly language level. The third part involves integrating the tagging scheme into a zero-kernel operating system (ZKOS). During this mapping, implementations and simulations we learned a few things that have not been discussed in the prior literature.

This poster provides an overview of the parts of this project, including the simulator, new tagging scheme and ZKOS.

II. CONCEPTS

Modern hardware tagging approaches have followed one of three main lines of research. The first is dynamic information flow tracking (DIFT) which adds taint bit(s) to user provided data, propagates those taint marks and throw security exceptions when the data is used in unacceptable ways (e.g., used as memory addresses to modify flow control – i.e., stack-based buffer overflows). The second involves semantic protection of program data by adding additional control bits to the data and then generating errors when data usage violates the expected semantics indicated by the control bits. For example these approaches have been used for uninitialized memory checks, or fat-pointers for bounds checking. The third approach uses tags to augment the separation provided by hardware protection rings, providing a much finer granularity of protection and a richer set of security domain tags.

In the literature we have also found that some of the tagging schemes are being implemented by hardware developers and the testing and simulation of those techniques is based on select, hand generated assembly code, possibly within circuit simulators. This places a great restriction on the completeness of the testing or the evaluation of the impact of the scheme in a complex software system. Other proposed schemes place an additional burden on the software developers – the compiler writer, the operating system developer or the application developer. The later is the developer we trust the least, since they are the ones developing vulnerable code in the first place and are not likely to understand the added complexity of new tagging schemes.

III. SIMULATOR

To provide a functional evaluation of proposed tagging schemes, and to allow for comparison, we modified the SPARC Instruction Simulator (SIS) that is included with GDB. We added hooks into the simulator, and developed a template for adding specific tagging scheme functionality – which includes programming of the tag engine, tag propagation and tag checking for the different instructions/instruction classes affected by the proposed schemes.

We have implemented several tagging schemes that we found in the literature, as well as a new scheme we developed. The schemes we have tested included those that implement DIFT, uninitialized memory checks as well as memory bounds checking. During our implementation, we noticed some

disconnects between what is reported in the literature, and what we were able to accomplish with the simulator. Some schemes require cooperation with the operating system (to taint the user input data) or compiler (to populate the fat pointers) in ways that are not obvious reading the published papers. We found one approach that claimed hardware tagging could be used to prevent SQL-injection attacks; only to determine that the solution was really a software-based solution (meant to replace a vulnerable software-based solution written by the same group of developers).

IV. A New TaggingG Scheme

As our research progressed, we developed a new tagging scheme to support fine-grain access control and to support implementation of a ZKOS. Our tags introduce the concepts of owners and code-space for data and code in the system. The owner portion of the tag reflects the end user of the data (or at least the subject of the data). For example, the data in the process control block for process 35 in the system is tagged with a process 35 owner). The code-space portion of the tag indicates the module that is authorized to manipulate the data. For example, the scheduler module is authorized to manipulate schedule related data in the process control block. We also provide portions of the tags to differentiate code from data, function entry points from regular executable code, and read/write access to memory. This can also be used to provide access control to specific functions, allowing for the creation of internal functions and isolating OS kernel modulus from the authorized user interface OS modules. We can include DIFT-style protection in the tags, and are in the process of examining the utility of that and other extensions.

Our first tagging model was specified using a 'C' programming language view of the world. When we went to implement the tags in the simulator we found some disconnects between the 'C' model of the world and the assembly language/microprocessor architecture view, which required some changes in the tagging scheme. We wrote many small test cases to evaluate many features of our schemes. For each case we would set appropriate tags, turn on the tag propagation and checking engine, and evaluate the results. We found that we were constantly thwarted by "optimization" features of the compiler that are not discussed in the related research.

V. Zero Kernel Operating System

A ZKOS differentiates itself from a normal operating system by acting as a run-time executive with security features. A run-time executive consists of a set of library routines that provide hardware abstractions, common services and system management for the user, all running in the user address space with the same permissions as the user. A ZKOS provides this close coupling of services to application code, reducing the need for costly context switches, but providing secure separation and access control through use of advanced hardware tagging features.

We have decided to take the RTEMS run-time executive and modify it to run on our modified tagging hardware as a ZKOS. We are currently part-way through the port, adding additional functionality to RTEMS to support multiple users and our tagging scheme (RTEMS currently supports a multi-threaded, multi-processor, single-user execution model). We are executing RTEMS both on our simulator and now on an FPGA implementation of the modified SPARC processor developed by a Cornell University research group.

VI. Lessons Learned

Initial work on security tagging architecture seemed interesting; we felt that we could give the hardware the ability to help us enforce security by providing fine-grain protection. We used the 'C' programming language as the initial model of execution, giving us a high-level language approach to the security model, while being able to reason about the lower-level security operations. When we moved to implementation, we found we had to look at how the assembly language implementation actually worked. We found several issues that were missed at the higher level, and in the discussions of other tagging schemes in the literature:

- Compiler optimizations can change and/or remove security relevant code. For example, we could tag a small function with a high-level security tag, hoping to prevent user access to the tag. The compiler can then "in-line" the function, effectively moving the code into user code space and ignoring the security tags. This was especially a problem when writing small test cases for evaluating the tagging and the simulator.

- Hardware features can change the execution and security model. The SPARC processor uses a register window to improve performance, and does not necessarily use the stack. Security models that assume access to the run-time stack may fail when the compiler does not implement a stack, but instead just uses the hardware features.

- History has shown us that programmers make mistakes. They will forget about adding security features, they will leave code vulnerable to attack or they will mislabel or misuse security tags. We have no clear indication that the addition of security tagging architecture will protect programmers from themselves, or that the added complexity of the tagging hardware control software will not make the security problem even worse.

VII. Conclusion

There is an ever increasing number of software developers building new applications, using new services, protocols and languages on hardware with ever advancing features and complexity. These developers are not all well trained in software engineering of secure systems, and often have a mental model disconnect between their view of the execution environment (as well as the end users) and reality. Worse yet, many of these developers have little formal training in software development, but rather took a few courses while they pursued an education in another discipline, but ended up being software developers.

We need enhanced hardware and software system security features that can help us protect the software developers from themselves, and simplify the development of more robust, more secure software. We believe hardware tagging may provide some of that help.