Korset: Automated, Zero False-Alarm Intrusion Detection for Linux

Ohad Ben-Cohen
Tel-Aviv University
ohad@bencohen.org

Avishai Wool
Tel-Aviv University
yash@acm.org

Abstract

Host-based Intrusion Detection Systems traditionally compare observable data to pre-constructed models of normal behavior. Such models can either be automatically learnt during a training session, or manually written by the user. Alas, the former technique suffers from false positives, and therefore repeatedly requires user intervention, while the latter technique is tedious and demanding.

In this paper we discuss how static analysis can be used to automatically construct a model of application behavior. We show that the derived model can prevent future or unknown code injection attacks (such as Buffer Overflows) with guaranteed zero false alarms. We present Korset, a Linux prototype that implements this approach, and focus on its Kernel implementation and performance.

1 Motivation

Throughout computer history, there is an on-going battle between attackers and defenders. Attackers expose and exploit application vulnerabilities on a daily basis, and as a result, software vendors regularly apply fixes to close breaches and mitigate attacks. Alas, it's a seemingly endless cycle that has attackers on the upper hand, as defenders are mostly responding.

Since even fully patched applications may have unknown security flaws, defenders commonly use Host-based Intrusion Detection Systems (HIDS's) in order to augment security. The advantage of using HIDS's relies on the fact that they may stop attacks which exploits an application vulnerability that is still publicly unknown (a.k.a zero-day exploit) or against which a patch is not yet provided.

HIDS's identify malicious activity typically by comparing a variety of observable data to a pre-constructed model of normal application behavior. When a running process deviates from its model of behavior, it is assumed to be subverted by an attacker. In such an event the HIDS can take actions to prevent the attacker from damaging the system, e.g., by terminating the hijacked process.

There are two classic methodologies for constructing an application's model of normal behavior. One prevalent methodology infers the model from statistical data: the model is constructed over a period of time, called "training", which is assumed to be attack-free (and hopefully typical). During the training period, the behavior of the application is observed, collected and transformed into a representative model. After the model is constructed and the training period is over, the HIDS monitors the process, and any deviation from the constructed model is considered an attack and may result in the termination of the process. This methodology is highly automated and capable of detecting a wide range of attacks, but since it is based on statistical data, it has the inherent problem of false positives. Recent methods have arrived at fairly low false positive rate, however in practice it is still a major problem.

A second common HIDS methodology for constructing an application model of normal behavior is based on generating application policies. Such policies define the allowed behavior of the program, using rules that precisely specify which system resources a process can access and in what way. The policies can be written either by the developer himself or by a knowledgeable user, because writing them requires a precise understanding of the expected behavior of the application. The advantage of using program policies relies on the fact that they can describe the program's behavior as accurately as the program code itself, and thus can completely eliminate false alarms. Alas, manually writing accurate program policies is not for the faint of heart as it is a tedious and demanding task.

An HIDS that would be able to automatically derive ac-

curate program policies will enjoy both worlds of zero false positives and automation. This can be achieved using static analysis methods as explained in section 2.

2 The General Idea

Korset's model of application behavior is Control Flow Graphs (CFG) induced from the source code and object files of the program. Assuming that the most practical ways for an attacker to inflict damage involve system calls, Korset prunes the CFGs from the nodes that do not represent system calls. The resulting model is an automaton that represents the legitimate order of system calls that an application may issue. This automaton is then enforced by Korset's monitoring agent, which is built into the Linux Kernel, by simulating every emitted system call. When a divergence from the automaton is encountered, the running process is terminated.

Assuming that the program was written with benign intent, and it isn't self-modifying, then its source code reflects the full extent of the legitimate application behavior, and nothing else. Every possible path of execution is obviously represented in the source, and as a result, also in the induced automaton. Every sequence of system calls, that does not match the derived automaton, couldn't have been issued by the program itself and therefore can be safely regarded as an intrusion. This leads to Korset guaranting zero false positives!.

3 Architecture

Korset has two main subsystems (see figure 1), described in the following subsections.

3.1 The static analyzer

The static analyser is a User Space subsystem that is responsible for creating the final application CFG. When the static analyser is enabled, the application CFG is automatically created as part of the compilation process. When the user builds an application (by running make, or compiling a random source file), the static analyser also creates the CFG. This is achieved by wrapping the GNU build tools (gcc, ld, as, ar), in a way that is transparent to the build system. As a result, a CFG is constructed for every built object file, executable or library (currently only static libraries are supported). The CFGs of C programs are initially derived using gcc's capability to dump a representation of the program's CFG during compilation, and the CFGs of Assembly programs are derived by analysing their object code (see figure 2). After the CFGs of the application's functions are created, they are linked together to create a unified CFG that corresponds with the executable of the application (see figure 3). Along the way, the graphs are simplified by tossing away CFG nodes that don't add relevant information, and as a result the final CFG is consisted exclusively of system calls nodes (see figure 4).

The CFG simplification is a lengthy process which includes numerous steps of graph determinizing and automaton manipulations. The complete theory and algorithms behind this process is described in length in our companion paper [BW08]. The end result, as can be seen in figures 5 and 6, is designed to achieve minimum run-time overhead: the final CFG is consisted of only system calls nodes and is actually a completely deterministic automaton. It is then transformed to a binary representation, which is designed to achieve maximum performance by placing the possible emitted system calls of each graph node closely after the location of the node itself (see figure 7).

3.2 The Monitoring Agent

Korset's monitoring agent is built natively into the Linux Kernel. When a monitored program is executed, the monitoring agent loads its CFG, observes the issued system calls by the process and validates their legitimacy by simulating them on the induced automaton.

Following is a description of the components that the monitoring agent is built of.

Note: support for the changes described in subsection 3.2.1 is added only if the Kernel's build variable `CONFIG_SECURITY_SYSCALL` is enabled. Likewise, support for all other changes described in subsections 3.2.2-3.2.5 is added only if the Kernel's build variable `CONFIG_SECURITY_KORSET` is enabled.

3.2.1 Linux Security Modules changes

Linux Security Modules (LSM) is a Linux Kernel security framework that provides, among other things, a set

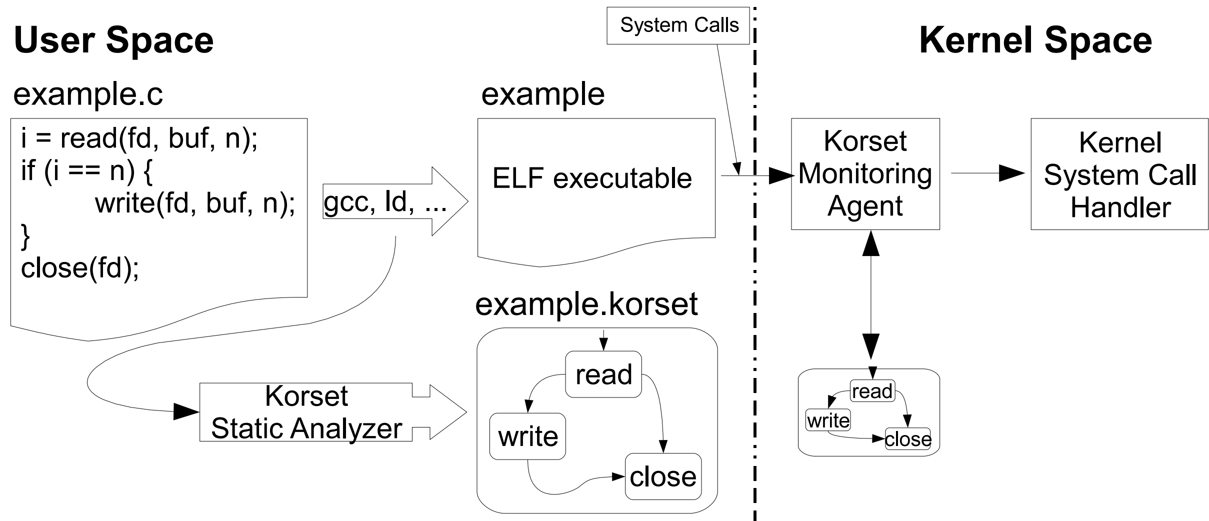


Figure 1: Korset's system architecture. On the left an application is compiled. Korset's static analyzer observe the building process, and creates a corresponding Korset graph. On the right, Korset's in-kernel monitoring agent loads the application's graph when it is executed, and then monitor the issued system calls by simulating the automaton. For simplicity, there is no notion of a process in the figure.

of security hooks that are used to perform access control. These hooks can be used by security modules to implement any desired model of security. In order to support the monitoring of system calls, we have added a new LSM hook, called `security_system_call`, into the `security_operations` structure (defined by `include/linux/security.h`). This new hook function is called every time User Space makes a request to execute a system call, by a handful of assembly instructions that we have added to the `system_call` handler (no support yet for the x86's sysenter facility and in general only the x86 architecture is currently supported). The `security_system_call` hook function is given two arguments:

1. The location of the `struct thread_info` of the current process (which have just made a request to execute a system call), retrieved from the process' Kernel stack using the `GET_THREAD_INFO` macro. The `struct thread_info` can be used to access the process' `task_struct` structure (the Linux process descriptor), where process security state is maintained by Korset.
2. The requested system call number, as given in `%eax` from User Space.

Any security module that registers the `security_system_call` hook should use these two arguments to decide whether or not to allow the execution of the desired system call. The hook function should return zero if permission to execute the system call is granted. If zero is returned, the system call handler continues with normal execution of the system call. Otherwise, the system call handler immediately returns to User Space with an `EACCES` error (Permission denied).

As with other LSM hooks, a security module who wishes to use the `security_system_call` hook should call `register_security` to set `security_ops` to refer to its own hook function.

3.2.2 task_struct changes

In order to maintain a per-process automaton in the kernel, we have added the following new fields to the `task_struct` structure:

- `korset_graph` - the location, in memory, of the automaton. When a process is not monitored, this field holds a NULL.
- `korset_node` - an offset, that together with `korset_graph`, yields the location in memory of the automaton node that the current process is at.

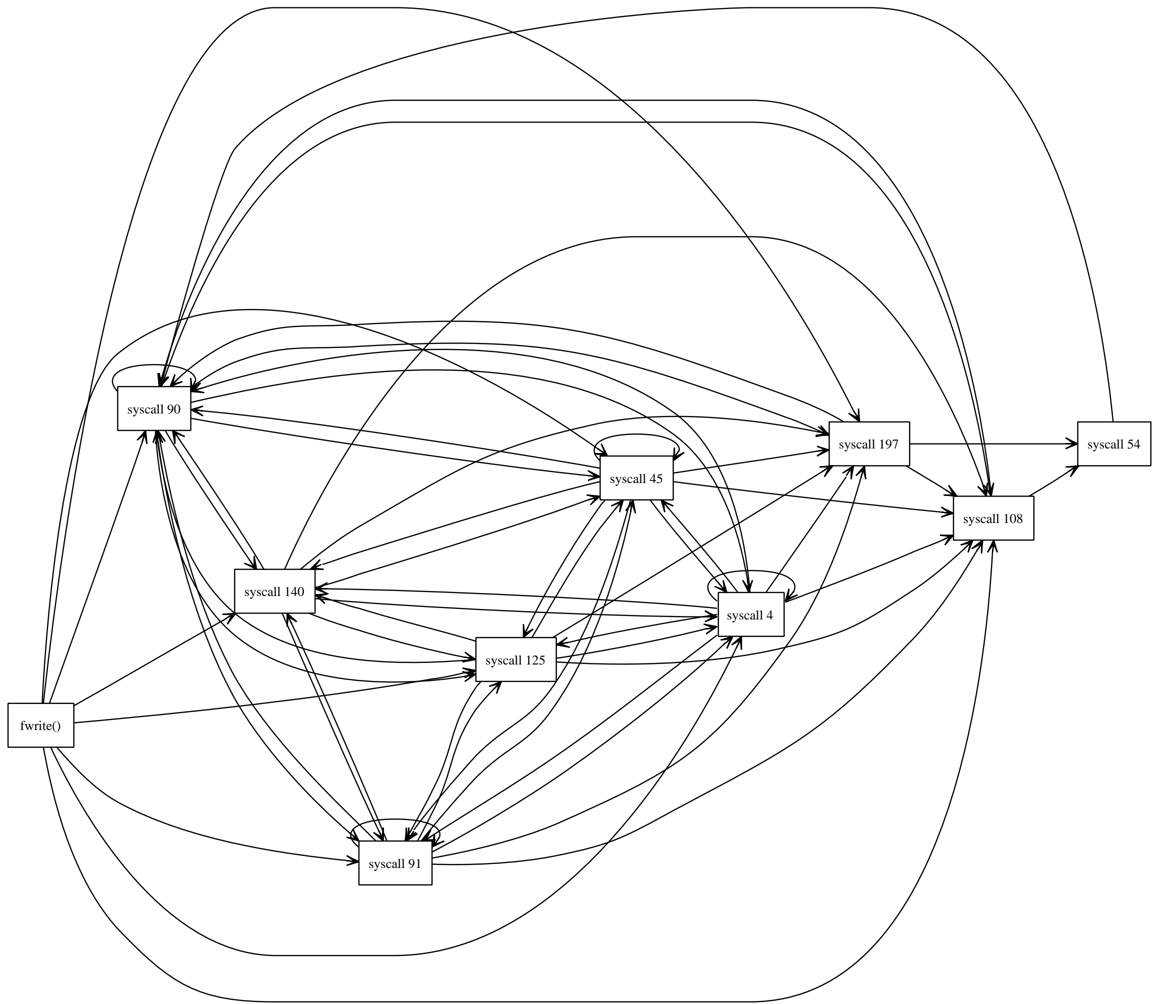


Figure 6: The CFG of glibc's fwrite(). The system calls involved are write(4), brk(45), ioctl(54), mmap(90), munmap(91), fstat(108), mprotect(125), _llseek(140) and fstat64(197).

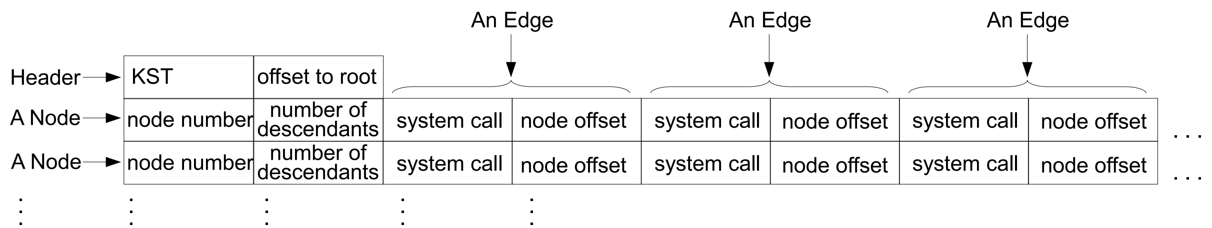


Figure 7: The binary representation of Korset's graphs is geared towards runtime efficiency

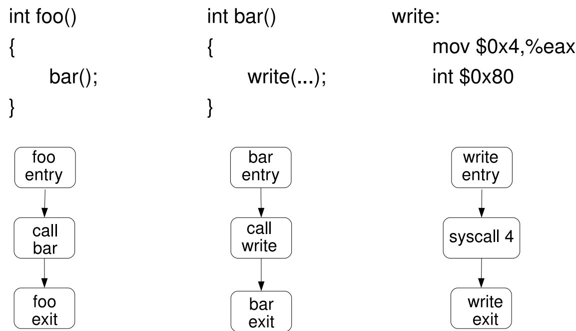


Figure 2: Three simple functions and their representative control flow graphs, as constructed by Korset. Note that `write`, which belongs to glibc, is a simple wrapper around a direct system call (Korset does not yet support the x86's sysenter facility).

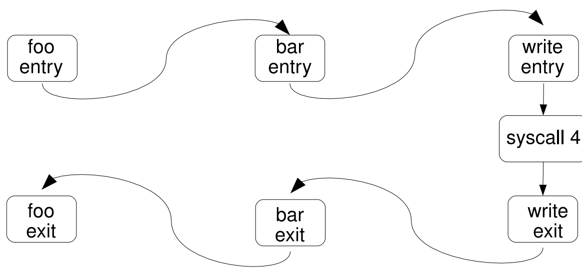


Figure 3: The CFGs are linked together in order to construct the final CFG of the program's executable.

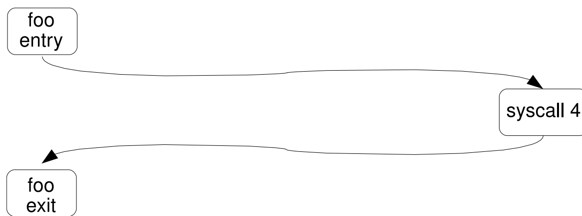
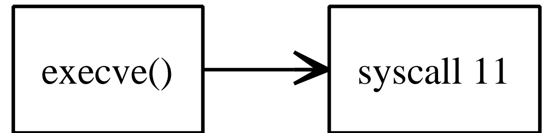
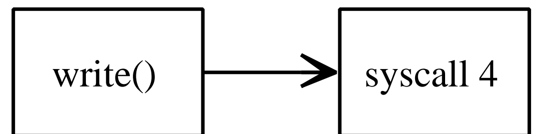
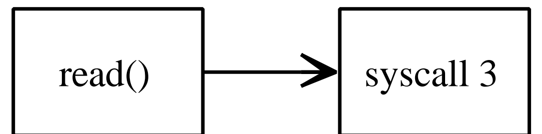


Figure 4: The resulting CFG is very simple since Korset prunes away all graph nodes that do not add relevant information







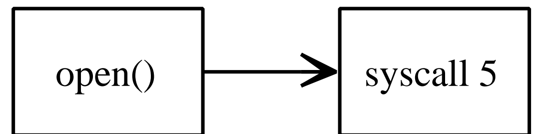


Figure 5: The CFGs of glibc's execve(), write(), read() and open(). These library routines are simple wrappers around the relevant system call, as reflected in the CFGs. The system calls involved are read(3), write(4), open(5) and execve(11).

- korset_size - the size, in bytes, of the automaton that is pointed to by korset_graph. Used by automaton sanity checks.

3.2.3 CFG Loader

When app is executed, Korset looks for the file app.korset, which, if exists, holds the CFG of app. This is done in fs/exec.c by the do_execve function, in almost exactly the same manner as the Kernel looks for app's executable itself (with the exception that only READ permissions/flags are needed instead of the usual EXEC ones). If the file app.korset exists, Korset loads it from disk. This is done in fs/binfmt_elf.c by the load_elf_binary function (currently only ELF executables are supported by Korset), again in a very similar way to how the Kernel loads the executable itself. After app's graph is successfully read from disk, its location in memory is put in the korset_graph field of the current process, the korset_size field is updated with the graph's size and the korset_node field is set to point to the graph's root node (the offset to the root node is taken from app.korset. see figure 7).

3.2.4 CFG Enforcer

Whenever a process issues a system call a, Korset's security_system_call LSM hook function is called. If korset_graph == NULL then the current process is not monitored, and a is immediately approved. Otherwise, if korset_graph != NULL, then the current process is monitored, and then Korset validates the legitimacy of a by simulating one step of the automaton. This is accomplished by checking the outgoing edges of the current node, one by one, looking for an edge that is carrying a. If such an edge is found, the korset_node field of the current process is updated to point to the destination node of the found edge, and a is executed. If such an edge is not found, the current process is terminated, and security_system_call returns -1 to the system_call handler. As a result, the system_call handler does not execute the requested system call. Instead, it immediately returns to User Space as explained in 3.2.1.

In some cases, after an edge carrying the requested system call is found, it is also desired to manipulate its location in the graph for performance reasons. This manipulation is done within the enforcing code. For more information see section 4.

3.2.5 CFG Dumper

As mentioned in the previous subsection, sometimes the application CFG is manipulated while it is enforced. This may happen for run-time optimization reasons, that are described in section 4. In those cases it might be desired that the graph file app.korset itself will be updated to reflect the changes. This is where Korset's CFG Dumper kicks in. When a process terminates, and the conditions described in section 4 are met, the updated CFG is dumped to disk. This is done in the do_exit function, defined by kernel/exit.c, in a manner that resembles the way the Kernel dumps a core file.

4 Runtime Optimization via Frequent Edges First

Validating a system call a involves matching a against each of the outgoing edges of the current node v, until either a match is found or all outgoing edges have been traversed. The time complexity of this operation is obviously $O(d)$, where d is the number of outgoing edges v has. The actual overhead is obviously dependent on the position, in memory, of the more frequent edges. If a system call a is carried by the first edge in memory, only one matching iteration will be performed to locate it, and the run-time overhead will be minimal. Based on that simple observation, we added the following Frequent Edges First (FEF) algorithm to Korset's monitoring agent:

1. Every time a system call match is made to an outgoing edge e of node v, e is moved-to-front, i.e., it is removed from its i-th location in memory, the $i-1$ edges that are located in positions $1..i-1$ are shifted to positions $2..i$, and e is placed in position 1.
2. When a monitored process finishes its execution, its updated CFG is dumped to disk.

This means that when the FEF is enabled, the CFG dynamically changes during the execution of the process. After the process terminates, the most frequent edges

of that specific execution are positioned before the less frequent ones. If the next execution of that application would have a similar system call sequences, Korset's overhead will be substantially smaller. The FEF can be used in two modes:

1. Always Enabled - in this mode the CFG will always be updated during process execution according to the FEF algorithm. The main benefit of this mode is that a recurring sequence of system calls will incur minimal overhead. However, this mode introduces additional computation that is performed per observed system call, which may actually increase Korset's run-time overhead. Therefore, we do not use this mode.

2. Only Once - in this mode the FEF is used as the final step of CFG construction: the relevant application is executed once, in a common workload. Upon termination, Korset dumps the updated CFG to disk, which is then used as the application's newly constructed CFG. This way, when a new CFG is constructed, the position of its outgoing edges reflect a real execution of its program rather than a random order. This mode is obviously cheap - it has no runtime cost. Its only cost is in the initial phase of CFG construction. Despite the low cost and obvious limitation of this mode, it is quite effective. We have found that even a single FEF adjustment of a CFG during a single execution of a program significantly improves Korset's performance in future executions.

The 'Always Enabled' mode can be further tuned to reduce run time cost (E.g., an outgoing edge can be advanced only if it is not in the first t locations, where t is a tunable variable that might be different for each application). This is a topic for further research. In contrast, the 'Only Once' mode is always used, since it has no run time cost, and it was found to be notably effective.

5 Run-Time Micro Benchmarks

Figure 8 demonstrates the percentage of overhead imposed by Korset's monitoring agent via micro benchmarks for four system calls with different speeds (write is the slowest while setuid is the fastest). For a single system call a, the actual overhead depends on the position in memory of the edge carrying a. We measured three different scenarios for each of the system calls:

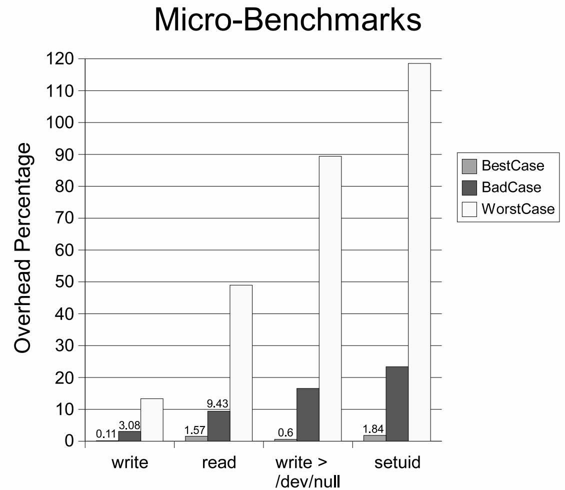


Figure 8: Micro-benchmarks of Korset's overhead on four system calls with variable speed. setuid is the fastest system call and thus has the biggest percentage of run-time overhead incurred by Korset.

1. Best Case: The matching edge is in the first position. While this is the best runtime performance that can be achieved, high precision models (i.e. with low branching factors), should produce similar results.

2. Bad Case: The matching edge is in the 50th position. This overhead level can only be achieved with models that have bad precision since a node with 50 or more outgoing edges is very limited with its constraining effectiveness.

3. Worst Case: The matching edge is in the 325th position (there are 325 different system calls in Linux 2.6.24). Measured for comparison.

The figure shows two things: (1) The overhead imposed by a small branching factor is negligible. Achieving a small branch factor is obviously a factor of precision, but the same performance can be achieved also by placing the more common graph edges first (which is what our Frequent Edges First process does). (2) As long as the system call itself is slow, Korset run-time penalty is small (in percentage) even if the branching factor is big. As the system call is faster, Korset runtime penalty is becoming more significant (in percentage). Measuring Korset's performace on simple IO-centric applications yielded results similar to the best case (around 1% overhead) [BW08].

6 Related Work

The idea of applying code-based static analysis techniques to automatically construct models with zero false alarms was introduced in a seminal work by Wagner and Dean [WD01]. Their work covers the theory behind the idea, and is a recommended read. Wagner and Dean used a non-deterministic model which resulted in big runtime overheads, and their prototype was implemented in Java. Continuing their work, Giffin et al. demonstrated static analysis of SPARC binary code to generate system calls CFGs of Solaris application [GJM02, HGH+04, GJM04, GDJ+05]. Giffin et al. have introduced numerous automaton manipulation techniques and binary modification techniques to increase model precision and reduce non-determinism. Methods to increase the model precision were introduced, based on additional observable data. More specifically, it was suggested to add system call arguments [WD01, GJM04], program counter and call stack information [HGH+04, GJM04] and environment information [GDJ+05]. A completely deterministic model, which results in better run-time performance, was suggested [BW08]. Lastly, the code-based static analysis model is not foolproof. An attacker can intentionally issue system calls in order to arrive to a desired automaton node. This type of attack, called `mimicry attack`, was introduced by Wagner et al. at [WD01, WS02].

7 Current Status and Future Work

Korset is still a very young prototype. It blocks real shellcodes, and it was found to incur negligible overhead on simple IO-centric applications (see our companion paper [BW08] for a detailed evaluation and analysis), but it is still very far from a mature HIDS. It does not yet support dynamically linked applications, multi-threaded applications, signals, `setjmp/longjmp` etc. In addition, there is still a lot of work required to increase the precision of the automatons, e.g., better assembly analysis, better indirect calls analysis, add additional observable data to the model, improve the automaton manipulation algorithms etc. Korset is hosted at `www.korset.org`.

8 Conclusion

Korset is an HIDS prototype that automatically constructs models of application behavior, and enforces them from the Linux Kernel with guaranteed zero false positives. Korset is capable of stopping future, or publicly unknown code injection attacks (e.g., buffer overflows). Although Korset is still a prototype, it demonstrates a viable HIDS methodology with promising intrusion detection properties.

References

[BW08] Ohad Ben-Cohen and Avishai Wool. Korset: Making intrusion detection via static analysis practical. Submitted for publication, 2008.

[GDJ+05] Jonathon T. Giffin, David Dagon, Somesh Jha, Wenke Lee, and Barton P. Miller. Environment-sensitive intrusion detection. In *Recent Advances in Intrusion Detection*, pages 185–206, 2005.

[GJM02] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, pages 61–79. USENIX Association, 2002.

[GJM04] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *Proc. 11th Annual Network and Distributed Systems Security Symposium (NDSS)*, 2004.

[HGH+04] H.H. Feng H.H., J.T. Giffin, Yong Huang, S. Jha, Wenke Lee, and B.P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings IEEE Symposium on Security and Privacy*, pages 194–208, 9-12 May 2004.

[WD01] David Wagner and Drew Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 156, Washington, DC, USA, 2001. IEEE Computer Society.

[WS02] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 255–264, New York, NY, USA, 2002. ACM.