Runtime Integrity Checking for Exploit Mitigation on Lightweight Embedded Devices

Matthias Neugschwandtner[1,2], Collin Mulliner[2], William Robertson[2], and Engin Kirda[2*]

[1] IBM Research Zurich, Switzerland
[2] Northeastern University, Boston

Abstract. Entering the age of the Internet of things, embedded devices are everywhere. They are built using common hardware such as RISC-based ARM and MIPS platforms, and lightweight open software components. Because of their limited resources, such systems often lack the protection mechanisms that have been introduced to the desktop and server world. In this paper, we present BINtegrity, a novel approach for exploit mitigation that is specifically tailored towards embedded systems that are based on the common RISC architecture. BINtegrity leverages architectural features of RISC CPUs to extract a combination of static and dynamic properties relevant to OS service requests from executables, and enforces them during runtime. Our technique borrows ideas from several areas including system call monitoring, static analysis, and code emulation, and combines them in a low-overhead fashion directly in the operating system kernel. We implemented BINtegrity for the Linux operating system. BINtegrity is practical, and restricts the ability of attackers to exploit generic memory corruption vulnerabilities in COTS binaries. In contrast to other approaches, BINtegrity does not require access to source code, binary modification, or application specific configuration such as policies. Our evaluation demonstrates that BINtegrity incurs a very low overhead – only 2%, – and shows that our approach mitigates both code injection and code reuse attacks.

1 Introduction

Modern embedded devices are built using common hardware such as RISC-based ARM and MIPS platforms and open software components such as Linux. Thanks to their relatively high spread, constant uptime and common components, embedded devices have become an attractive target for attackers. Routers, in particular, have been regularly abused as hosts for botnets in the past years [3,18,19]. Similar to traditional desktop and server environments, embedded systems are attacked mostly through exploitation of software vulnerabilities such as memory corruptions.

Memory corruption vulnerabilities and attacks that exploit them remain one of the major issues in computer security [17,22]. There is an ongoing arms race as defenders build new mitigations, and attackers discover ways to bypass these defenses. For example, Data Execution Prevention (DEP) was created to defend against code injection attacks. However, attack techniques such as generic code reuse (e.g., return-to-libc)

[*] Thanks to Secure Business Austria

and return-oriented-programming (ROP) were then developed that allowed attacks to be launched without the need to inject code. As a remedy for code reuse attacks, Address Space Layout Randomization (ASLR) was proposed. However, ASLR can also be bypassed if the attacker can leak addresses from a target process. Recent work [4,13] started using hardware features to mitigate ROP attacks on desktop computers and servers running on the x86 platform.

Note that previous advancements on exploit mitigation focused mainly on desktop computers and servers, and have neglected securing embedded devices against software vulnerabilities. There is even a gap within the embedded systems world where CPUs for high-end applications contain security features such as DEP while the (cheaper and more common) low-end versions do not and thus leave the majority of embedded devices vulnerable to even simple attacks that have been eradicated in the desktop and server world.

This paper presents, BINtegrity, a practical, low-overhead system to mitigate memory corruption attacks that is specifically tailored towards embedded systems. Our approach aims to protect against state of the art attacks while still being practical. We specifically target platforms based on RISC architectures and systems where we only have access to program binaries.

The fundamental insight we gained from looking at previous work (Section 2) is the need to combine multiple techniques and aspects in a novel way to practically mitigate exploitation attempts. BINtegrity ensures that system calls are only invoked in a legitimate way by checking the runtime integrity of a program. Integrity means that the runtime state of a program has to be coherent with its binary executable. Coherence is given if the following conditions are met: *i*) the invoked system call is actually used by the binary *ii*) the system call arguments match the ones specified in the binary *iii*) the observed call chain is reflected by the binary.

Our approach (Section 4 and 5) leverages several common design features found on all RISC CPUs to reconstruct part of the call chain and extract properties from a binary executable in an efficient way. To the best of our knowledge, we are the first to leverage these fundamental architectural features of RISC CPUs to build a security system. Our method is lightweight and allows our system to execute all operations at runtime.

BINtegrity is designed to be transparent to legacy binary code and works for arbitrary programs without requiring access to source code, prior training phases, or binary modifications. Note that previous approaches (e.g., intrusion detection based on system call monitoring and full control flow integrity) do not offer similar flexibility and are, thus, not practical for embedded systems. The only requirement for our approach is the presence of a kernel-based runtime component that – on-demand – extracts properties directly from program binaries and checks them against runtime behavior. Our technique guarantees that only system calls that are actually used by a program can ever be invoked by a corresponding process. BINtegrity enforces strong restrictions on *how* system calls can be invoked. When combined, these two features significantly reduce the attack surface available to code reuse attacks, and vastly limit the options for successful code injection attacks.

We built a prototype implementation (Section 6) of BINtegrity as an extension to the Linux operating system for the ARM and MIPS architectures and evaluated it on two common and popular off-the-shelf devices: a MIPS-based router, and an ARM-based smartphone. Our evaluation (Section 7) demonstrates that BINtegrity effectively mitigates code injection and code reuse attacks while incurring an extremely low performance overhead of only 2%.

The paper makes the following contributions:

– We present a new practical method to efficiently extract and leverage process runtime information on RISC architectures. The extracted information is used to create and check properties against the program's binary file on disk.
– We propose a lightweight technique for restricting memory corruption attacks for COTS binary programs. Our system is the first to specifically target the RISC architecture leveraging some of its unique features.
– We developed a prototype implementation of this technique called BINtegrity as an extension to the Linux operating system for the ARM and MIPS architectures. The source code of BINtegrity is publicly available at `www.bintegrity.org`.
– We evaluated BINtegrity on two real-world systems, and show that our technique effectively defends against code injection and reuse attacks. We also show that it incurs a low performance overhead and, therefore, represents a practical, generic defense for embedded systems.

2 State of the Art

Mitigation of memory corruption attacks is a well-studied problem in literature. A wide range of approaches tackle the problem from various angles and at different stages during exploitation. Since preventing memory corruption in the first place is fairly difficult, most approaches focus on hindering an attacker in successful exploitation of a memory corruption vulnerability. After having corrupted the memory, the exploit has to redirect the control flow to either plain shellcode or a series of ROP gadgets. Basic defenses try to prevent shellcode injection or finding the desired code in memory while more advanced techniques such as Control Flow Integrity (CFI) aim at preventing malicious control flow redirection. Finally, an exploit has to invoke system calls to use functionality beyond pure computation. This fact led to a large body of research in the area of system call monitoring to detect malicious system calls.

Basic Defenses Data execution prevention (DEP) [2], prohibits instruction fetch from data-only memory regions. While DEP proved to be an effective mitigation of straightforward code injection exploits, it relies on hardware support that is not necessarily available on embedded devices, depriving them of the benefits of this basic defense technique. Address space layout randomization (ASLR) places the text segments of a process at random memory locations. This probabilistic technique hinders the attacker in determining target addresses for code reuse attacks. The drawback of this technique is that unless the executable itself is compiled as a so-called PIE, randomization is limited to the dynamically linked libraries. In addition, the effectiveness of ASLR is limited on 32bit platforms. Current deployments of ASLR also only perform randomization at program startup, which makes it less effective with long-running applications.

Mitigating Code Pointer Corruption Code-Pointer Integrity [11] hides code pointers from being accessed by an attacker by storing them in a "safe", inaccessible memory region. While CPS, a relaxed variant of their solution, has relatively low overhead, it requires access to source code and is ineffective against attacks that do not require code pointer modifications [15].

Control Flow Integrity The property of CFI [1] describes whether a program's execution flow has deviated from its intended path. Provided that it can be both measured and enforced to full extent, attacks based on control flow hijacking could be completely prevented. The problem with CFI is that solutions that can operate on binaries [25] have been shown to be too coarse grained to be effective [8]. Recent approaches use context-sensitive CFGs for higher precision [21], but achieve low-performance overhead only by relying on hardware features currently not present on embedded devices [6].

System Call Monitoring and Policy Enforcement Early approaches on exploit mitigation entirely focused on the inspection of system call properties. Starting from writing policies for system call execution [14,20,23], these approaches eventually evolved to full-fledged mandatory access control systems such as SELinux[3] or AppArmor[4] that require a corresponding configuration effort.

A different line of work tries to detect anomalous system calls that would deviate from normal program behavior [10,7]. These systems rely on a runtime learning phase to model normal behavior. However, mimicry attacks [24] that hide their malicious system calls in a valid sequence or change the parameters to system calls have soon shown the limitations of such approaches. Recent work [16] claims to be resilient against mimicry attacks by working on extremely long execution paths.

3 Threat Model

Our threat model covers memory corruption exploits such as buffer overflows on binary programs running on the device. We assume that the attacker has access to the target application, and that she can launch local or remote attacks against it. We note that in order to have an effect on the system, the attack code has to request operating system services at some point, i.e. perform a system call. Without being able to perform a system call, its possibilities are limited to pure computational tasks while operations like file or network I/O as well as process control are impossible.
In summary, we assume that:

- The operating system kernel and the underlying hardware are trusted, and have not been compromised by the attacker.
- The application binary on disk is trusted, and cannot be modified by the attacker. We assume the same holds true for the shared libraries used by the application.
- The process memory is *untrusted*, since the attacks we mitigate are based on memory corruption. We assume that the attacker might be able to execute a limited number of instructions without being detected. Also, we assume that the attacker has the ability to modify and overwrite arbitrary memory locations, including regions that contain executable code.

[3] SELinux `http://www.selinuxproject.org`
[4] AppArmor `http://apparmor.net`

Based on our threat model, an attacker can manipulate the control flow of a process to the attacker's benefit through targeted memory manipulation. We do not cover memory corruption attacks that only change program data (e.g. file names in a write operation) and do not redirect control flow. This allows for two classes of attacks: *code injection* and *code reuse*. In case of code injection, the attacker introduces completely new code into memory and redirects control flow to it. Code reuse attacks on the other hand leverage code already existing in a process. They can operate at different levels of granularity, either targeting whole library functions (commonly referred to as "return-to-library") or small code chunks of the program that are stitched together (i.e., "gadgets").

4 Approach

Our goal is to create a *practical*, *efficient* and *effective* defense to mitigate exploitation of memory corruption attacks on RISC platforms for embedded systems.

We designed BINtegrity around the fact that code that has an effect on the system does so by making use of the operating system through system calls. The same principle is true for exploits. Hence, our approach is centered around system calls. Whenever a system call is invoked, we ensure that this is done in a legitimate way by checking the integrity of a program's runtime state. To this end, we extract several properties from the program's executable on disk that are relevant to the system call invocation at hand. We then compare these properties with the actual runtime state. In case the comparison succeeds, the program is allowed to perform the system call. Figure 1 shows a high-level overview of this process.

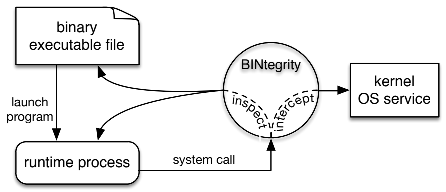


Fig. 1: A high-level overview of our approach. BINtegrity intercepts system calls and inspects both the runtime state and the binary executable image of a process. In case it detects a mismatch, the process is terminated.

The properties used in the comparison are extracted from the corresponding binary executable. Based on the origin of the last two items in the call stack before the system call invocation, we first check the existence of corresponding control flow instructions at the respective locations in the binary. We proceed by evaluating the targets of these control flow instructions and ensure that they match the call chain. As a next step, we compare the arguments of the system call with the instructions in the binary that precede the invocation and ensure that they do not contradict the actual arguments. Finally, we evaluate the import tables of the executables corresponding to the call stack, and ensure that all required import-export dependencies are indeed met. In case the runtime state's integrity has been violated, i.e. a mismatch is detected, the program is terminated.

The integrity checks based on these properties can effectively mitigate code injection and reuse exploits. To begin with, system call requests coming from injected code are predestined to originate from an unusual code location. Mitigating such attacks by checking code origins forces attacks to be constructed using only code reuse methods such as return-oriented-programming (ROP). In theory return-oriented programs are Turing-complete, but real-world ROP-based exploits are harder to construct since they require a significant higher skill level and more time. An attack that reuses existing code naturally comes from the right origin, but will have to differ in its control flow and function arguments to use system calls in a way that deviates from their regular invocation. By enforcing properties of the original code, BINtegrity restricts this most important step of code reuse attacks, thus effectively mitigating code reuse attacks.

For our design, we leverage common properties of RISC architectures. Specifically, we leverage the fixed-length instruction set to implement static analysis and code emulation in a very efficient way. Further, we utilize registers that are used during control flow transfer to efficiently construct a program's call chain. In contrast to related work, we do not rely on hardware features specific to certain processor families [13,4,26].

BINtegrity is transparent to existing applications, and can handle arbitrary binary programs. As a consequence, it does neither require access to source code nor binary instrumentation or any other pre-analysis stage. Finally, the on-demand fashion of BINtegrity ensures that it only causes overhead whenever a system call is actually invoked, making it applicable to existing device and software configurations.

Looking from a different angle, our approach provides functionality similar to policy enforcement systems such as SELinux and AppArmor. In contrast, though, BINtegrity provides this functionality implicitly as it does not require any policy configuration – restrictions are automatically derived from the binary.

5 Ensuring Runtime Integrity

In this section, we present how we check and enforce adherence of a program's runtime execution to its executable image with respect to system calls. The technique executes completely at runtime as a kernel extension for arbitrary program binaries on any RISC architecture that supports the equivalent of a link register. Ensuring the integrity comprises four steps: *i*) identification of a *trusted application code base*; *ii*) extraction of the *runtime state* at the time of a system call invocation; *iii*) *invariant extraction* from the original executable image given the runtime state; and, *iv*) *invariant enforcement* to ensure integrity. Figure 2 depicts an overview of this technique. In the following, we describe each of its components.

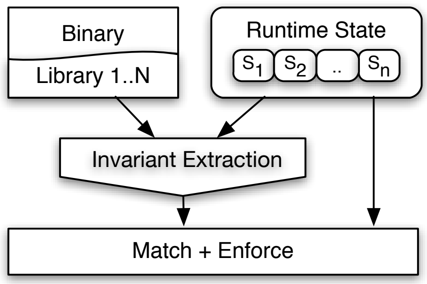


Fig. 2: Diagram depicting a high-level overview of ensuring runtime integrity. The key idea is to enforce security restrictions by matching a process' runtime state against invariants extracted from both the runtime state and facts from the corresponding executable image.

5.1 Trusted Application Code Base

The first requirement to ensure runtime integrity is to define a trusted application code base (TACB). This TACB refers to the program text of an application to be protected. The TACB is defined when a process is created by first allowing the runtime linker (e.g., ld.so on Linux) to load the executable image and any shared libraries it depends on into the process address space. At the point when control is transferred to the program itself, all mapped memory segments marked as executable – i.e., program text from the executable image and shared libraries – is taken as comprising the TACB.

5.2 Process Runtime State Extraction

Once a TACB has been established, the main part of the technique occurs at the time a system call is invoked. When a program invokes a system call and control transfers to the kernel, a runtime state is extracted from the process. This runtime state consists of the following information:

- The *return address* of the system call, which points to the successor instruction of the program call site. We denote it as ret_{sc}.
- The *link address*, which refers to the value that is stored in the LR (ARM) or RA (MIPS) register. This points to the successor instruction of the enclosing procedure call site before the system call invocation. We denoted it as ret_{lr}.
- The register that contains the system call number.
- All registers containing system call arguments.
- On MIPS, the jump target register.

Taken together, this state provides full information about the system call that is to be performed, its arguments, and a call chain of depth 2 in the program.

5.3 Invariant Extraction

Given a state, the next step of the technique is to extract invariants. These invariants are recovered by performing a lightweight static analysis of the program code referenced by the return and link addresses. We distinguish between two classes of actions performed during this procedure: *i*) *code invariant extraction* refers to analysis of the executed instructions leading to the invocation of the system call, while *ii*) *symbol invariant extraction* refers to resolution of the symbols for the last two entries in the program call chain.

Code Invariant Extraction To derive control flow information and invariants, code invariant extraction uses a combination of static analysis and lightweight execution emulation. First, BINtegrity performs backwards disassembly from the offsets into the binary given by $ret_{(\cdot)}$. Disassembly continues until either a control flow transfer instruction or a function prologue is found for the enclosing function $F(ret_{(\cdot)})$. Starting from the point where backwards disassembly has stopped, BINtegrity emulates instruction execution. As an execution state, we use a lightweight abstraction consisting solely of registers. We note that on RISC architectures that traditionally provide a large number of registers, we did not find the lack of a memory abstraction to impair the effectiveness of this approach. Focusing on registers also limits the number of instructions that must be supported. Once the execution reaches $ret_{(\cdot)}$, the current emulation state is collected; concrete values in this state are taken as state invariants. We also add the kind of control flow instruction that precedes $ret_{(\cdot)}$, and attempt to derive its target in case it is an indirect call.

Symbol Invariant Extraction Every program that uses an external library function needs some means to refer to that function. Executables refer to the functions they offer or use by means of symbols $S(F)$ encoded as simple character strings. The set of required functions are referred to as *imports* $IM(E)$ of an executable E, while the set of available functions are the *exports* $EX(E)$. After loading the executable objects into memory, the linker matches all exports against the imports and resolves the symbols to actual addresses. Symbol invariant extraction resolves the symbol $S(F(ret_{(\cdot)}))$ in the binary executable associated with $ret_{(\cdot)}$ and looks up whether it is exported or imported by this binary. It then adds this information to the set of collected invariants.

5.4 Invariant Enforcement

Given the extracted runtime state and collected invariants, the final step is to check the collected invariants against the state. That is, the approach checks for contradictions that indicate violation of a safety condition. In particular, the technique checks the following properties:

1. Code provenance
2. Call chain integrity
3. System call argument integrity
4. Symbol integrity

For a runtime state to be accepted as safe, all of these properties need to be verified successfully. In the following, we describe each of them in detail.

Code Provenance Code provenance enforces that only code from the TACB invokes system calls or their wrappers. Checks against the TACB are straightforward: both the system call *return address* as well as the *link address* have to point to code contained in the TACB to succeed – i.e., $ret_{sc} \in TACB \wedge ret_{lr} \in TACB$. The *link address* will either point to a location in the application binary, or to a location in one of the libraries that are used by the application. TACB checks are fast, as they do not require examination of the binary.

Call Chain Integrity To check call chain integrity on ret_{sc}, we verify whether the preceding instruction $Pred(ret_{sc})$ is indeed an instruction that invokes a system call. Figure 3a shows the system call wrapper for the `write` system call, with the system call return address pointing to ret_{sc}. In addition, we compare the system call number that is stored in a dedicated register of the state to check whether the correct system call handler has been invoked. For call chain integrity on ret_{lr}, we check that $Pred(ret_{lr})$ is a branch instruction to begin with. Depending on whether the emulation step provides us with the target of the branch, we also ensure that only the corresponding function $F(ret_{sc})$ is called by $Pred(ret_{lr})$. In Figure 3b this corresponds to the `bl` instruction calling the wrapper for `write`.

Argument Integrity Argument integrity enforces that the parameters of a system call invocation from the runtime state matches the results of the emulation. Of course, this can only be enforced if invariants are recovered for those registers – that is, an assignment derived from a constant value must have been observed during emulation. A further requirement for performing this match is that the parameters are not changed by a system call wrapper $F(ret_{sc})$. Section 5.5 describes argument integrity in more detail.

Symbol Integrity Symbol checks against the mapping established by the import and export tables of the code in the TACB prevent unauthorized use of functions. The intuition is that if a return address $ret_{(\cdot)}$ falls into the address range of an exported function of an executable, the symbol of the function has to be imported by some other executable. More formally, given a call stack of depth k with return addresses $ret_k, ret_{k-1}, \ldots$, two consecutive return addresses that point to different executables have to be linked by their exports, respectively imports.

$$\left.\begin{array}{c} ret_{k-j} \in E_a \wedge \\ ret_{k-j-1} \in E_b \wedge \\ E_a \neq E_b \end{array}\right\} \Rightarrow S(F(ret_{k-j})) \in \begin{array}{c} EX(E_a) \\ IM(E_b) \end{array} \tag{1}$$

Furthermore, if ret_{k-j} is known and $S(F(ret_{k-j})$ is exported, but ret_{k-j-1} is unknown, $S(F(ret_{k-j}))$ has to be imported by an executable that is not E_a.

$$S(F(ret_{k-j})) \in EX(E_a) \Rightarrow S(F(ret_{k-j})) \in IM(x), x \neq E_a \tag{2}$$

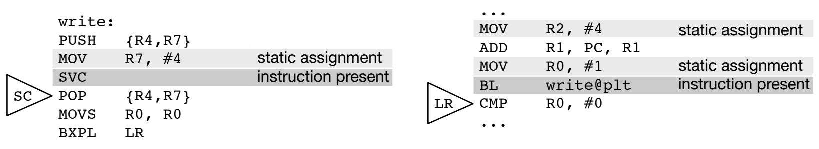
The runtime state provides us with ret_{sc} and ret_{lr}, which are equivalent to the last two entries in the call chain before the system call invocations, ret_k and ret_{k-1}. In the typical case, ret_k will point to a system call wrapper in a library E_a, which is imported by and called from the main program executable E_b, with $ret_{k-1} \in E_b$. This allows us to enforce a strong match between two executables E_a and E_b based on Eq. 1. If this implication holds, we continue with inspecting ret_{lr}. If it points to the main executable, there is nothing more to check. However, E_b might also be yet another library that provides some higher-level functionality. In case $F(ret_{lr})$ is exported as well, we can check it against the imports of all other mapped executables based on Eq. 2. In theory this could be an issue if $F(ret_{lr})$ is both exported and used internally by E_b, but not imported by any other executable. However, we did not encounter such a case in practice.

5.5 Function Call Arguments

A critical component of the execution emulation is deriving invariants on system call arguments. The specific idea is based on the observation that parameters to system calls are often composed of both static and dynamic values. Dynamic values are often memory addresses, while static values often specify length values and flags. By extracting statically assigned arguments from the application binary and comparing them against observed values, it is possible to infer whether execution actually proceeded along the expected path through the binary to the system call invocation.

Figure 3 shows the instructions that are involved in a call to `write`. Figure 3a shows the system call wrapper in the libc (bionic) and Figure 3b shows a call to `write` carried out in the application binary. The `write` system call takes three arguments. In our example, the first and third arguments are assigned statically, while the second argument is based on the content of another register. The second argument of `write` points to a buffer that holds the data that is to be written, while the first and third argument are a file descriptor (in this case, standard out) and the number of bytes that should be written.

During execution emulation we record static values for the first and third function argument because they were specified as constants in the executable image. These then become invariants that can be checked when that particular system call at that location in the executable is invoked. As a result no code reuse attack could abuse this `write` call to modify an arbitrary file. The second argument is marked as do-not-compare, as its value cannot be determined.



```
write:
PUSH    {R4,R7}
MOV     R7, #4          static assignment
SVC                     instruction present
POP     {R4,R7}
MOVS    R0, R0
BXPL    LR
```

```
...
MOV     R2, #4          static assignment
ADD     R1, PC, R1
MOV     R0, #1          static assignment
BL      write@plt       instruction present
CMP     R0, #0
...
```

(a) System call wrapper for the `write` system call. From BINtegrity's point of view, the system call return address points to the successor of the system call instruction.

(b) Call to `write` in the application binary. From BINtegrity's point of view, the link return address points to the successor of the procedure call instruction.

Fig. 3: Example of critical code regions that are analyzed at runtime by our system.

Some system call wrappers perform more operations besides invoking the system call. If these additional operations include modifications of arguments passed from the application to the system call, our argument integrity check would fail. Our system takes this behavior into account, and analyzes the instructions preceding the system call instruction to determine if argument registers are modified. One example of a system call wrapper that modifies arguments before invoking the actual system call is the wrapper for `open64` in uClibc compiled for MIPS. This wrapper modifies the second argument before invoking the system call by applying a bitmask to the second argument. In this case, our static analysis concludes that the second argument cannot be matched against the runtime state even if we are able to determine a statically assigned value in the application binary. The information about what arguments are modified by a system call wrapper is stored as part of the execution emulation procedure.

5.6 Dynamic Code Loading

While the TACB often does not change during execution, some processes do make use of interfaces like `dlopen` to dynamically load code. For this interface, an application can call `dlopen` to load a library, `dlsym` to resolve a function exported by that library, and invoke that function as any other. Internally, `dlopen` uses `mmap` and a small number of other system calls to load the library code into memory. BINtegrity handles dynamic code loading by tracking uses of `mmap`. Each time `mmap` is invoked by a process, our technique checks whether `dlopen` was called prior to calling `mmap`. This check is possible because of the invariant extraction mechanism we described above. If it is determined that `mmap` was executed on behalf of the `dlopen` function, the dynamically loaded library is added to the TACB.

6 The BINtegrity System

In this section, we present the BINtegrity prototype implementation, which is publicly available at `www.bintegrity.org`. BINtegrity is written as an extension to the Linux kernel, and is activated every time a process invokes a system call. The rationale be-

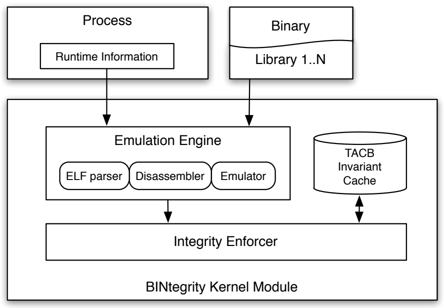


Fig. 4: Overview of the BINtegrity system.

hind this approach is that a program must always resort to services offered by the kernel to perform security critical actions such as file and network I/O, or to spawn new processes. Therefore, the system call boundary is an ideal location to collect process state information, and to enforce security policies.

6.1 Implementation

We developed BINtegrity as a loadable kernel module that is compatible with the Linux 3.x series. Since BINtegrity's operation has to be completely transparent to user mode programs, we do not introduce new techniques or modify existing ways to interact with the kernel. With support for different architectures in mind, we kept platform-specific code at a necessary minimum. That is, only components that require assembly-level support are platform-specific, and are easily ported to new architecture. Figure 4 shows an overview of BINtegrity's design, which is composed of the following.

Emulation Engine The emulation engine contains all components that extract invariants from the program's runtime state and executable image on disk. This includes a parser for ELF files to infer symbol information, as well as a disassembler and instruction emulator for both the ARM and MIPS instruction set.

TACB and Invariant Cache The per-process information BINtegrity maintains consists of the TACB and the invariant cache. The TACB keeps track of a process' memory pages that contain executable code, and serves as a reference when mapping addresses to binary files on the disk. The process of invariant extraction includes file parsing, disassembly, and instruction emulation – all relatively expensive operations. Therefore, we added a caching mechanism to our system that stores extracted invariants, thus limiting the number of actual extraction operations executed per process.

Integrity Enforcer The integrity enforcer performs invariant checks and acts upon their result. It is a lightweight component, as it is the most frequently invoked part of BINtegrity. After checking code provenance, this component queries the invariant cache. If it does not contain the invariants for the current state, it invokes the emulation engine to produce them. Subsequently, the invariant checks are performed. In case any of them fail, the process is terminated.

System Call Interposition System call interposition has been implemented as detour trampolines that invoke BINtegrity's main functionality before the system call handler is executed. BINtegrity intercepts execution immediately after the kernel's dispatcher has performed basic context-switch duties. As soon as all checks have passed successfully, the execution of the system call handler routine commences as usual. In case one of the checks fails, the task that issued the system call is terminated.

Disassembler and Emulator The disassemblers in BINtegrity operate in a linear sweep fashion. They support the instruction sets of the MIPS32 and ARM architectures, including the Thumb and Thumb2 instruction sets. The instruction set supported by the code emulator is reduced to the subset that operates on registers and immediate values.

6.2 Integrity Checking Levels

In order to improve performance, BINtegrity only interposes on a subset of the system call interface. Some system calls are invoked more frequently and thus more sensitive to enforcement overhead. Furthermore, system calls such as `write` are used by virtually every program and therefore do not require symbol checks. In contrast, system calls such as `execve` are rarely called and thus heavyweight checking can be performed without an observable performance impact.

BINtegrity therefore implements three distinct integrity checking levels, with each level adding additional checks to the previous level. The lowest checking level is the TACB check. Here, we only perform code provenance checks for the *return address* and *link address* using our TACB. The second checking level adds the checks for call chain integrity as well as system call argument integrity. The third checking level adds symbol checking. Naturally, the checking level directly affects which invariants need to be extracted. For instance, if symbol checking is not enabled, less extensive file parsing has to be performed.

In our prototype, we selected 33 security critical system calls that are used by BINtegrity. Of these 33 system calls, we configured 11 for checking level 2, and 22 for checking level 3. Checking level 1 is not used by itself. The list of system calls we perform checks on are listed in Table 2. We note that assigning checking levels to system calls is the only configuration our systems needs.

6.3 Invariant Caching

Emulated execution requires a considerable effort: reading files from the disk, parsing ELF information, disassembling machine code, and actually emulating execution. These operations would incur a severe performance overhead if they were executed for every system call invocation. Therefore, we implemented a caching mechanism that stores extracted invariants such that they can be reused in an efficient way during enforcement.

In particular, invariants are cached for the *return address* and *link address* individually. Splitting up the caching improves memory efficiency as system call wrappers are likely called from many program locations, while the system calls themselves are usually only invoked from a comparably small number of locations.

7 Evaluation

In this section, we evaluate the security characteristics and performance of BINtegrity. For the performance, we both take a look at BINtegrity's internal workings as well as application-level benchmarks.

7.1 Security Evaluation

In the following, we describe why BINtegrity effectively defends against both code injection and code reuse attacks.

Code Injection BINtegrity mitigates code injection attacks with early steps in the enforcement process. *Code provenance* checks of return and link addresses against the TACB prevent attackers from invoking system calls or their library wrappers from untrusted memory regions. *Call chain integrity* checking prevents the attacker from invoking system calls or library wrappers from memory areas that contain the application and library binary code, but have been overwritten with other code.

Code Reuse BINtegrity restricts the capabilities of code reuse attacks from different angles. *Call chain integrity* checking defends against abusing indirect jump targets to invoke library wrappers. *System call argument integrity* checking prevents the attacker from manipulating arguments of code that invokes library wrappers with static function arguments. *Symbol integrity* checking prevents calls to library functions that are not used by the application binary itself.

Table 1 shows an overview of the attacks covered by BINtegrity compared to other defense mechanisms. While we do not prevent code reuse, we greatly limit code reuse attack capabilities to resort to library functions or system calls. In summary, we provide a more fine grained protection against code injection than both DEP (which relies on hardware features not available on many RISC platforms) as well as approaches that strictly enforce write-or-executable memory pages such as grsecurity[5]. In addition, we provide a restricted form of control flow integrity at the system call boundary without the need for recompilation of the source code or rewriting the binary, and can achieve this with less performance overhead.

Attack	DEP	ASLR	LBR	CFI	BINtegrity
Injection	✓(DS)			✓	✓
Reuse		✓(L)	✓(G)	✓	(✓)

Table 1: Attacks handled by various protection mechanisms. DS = data segment only, L = library only, G = gadget only

Case Studies OSVDB-86824 describes a buffer overflow vulnerability on the D-Link DIR-605L router. The router's web server that handles login data processes user-supplied POST data without sufficient checks. Hence, remote exploitation is possible, leading to full system compromise as the web server runs with root privileges. Proof-of-concept exploits [5] inject shellcode on the stack that spawns a remote shell. BINtegrity's code provenance checks detect and prohibit such behavior.

As another example, the recent CVE-2013-4659 describes multiple buffer overflow vulnerabilities on the ASUS RT-AC66U router. In particular, the ACSD service's command processing routine is vulnerable and, again, can be used to completely compromise the router as the service runs with root privileges. Advanced proof-of-concept exploits [9] use code reuse techniques to first flush the data cache and subsequently invoke

[5] GRsecurity `http://grsecurity.net`

the `system` function in libc. As the latter function is not imported by the ACSD service binary, BINtegrity's symbol integrity enforcement effectively prevents this attack.

7.2 Performance

To evaluate BINtegrity's performance, we deployed it on a Samsung Galaxy Nexus (ARM) running Android and a Buffalo WZR-HP-G450H (MIPS) running OpenWRT. All results were obtained using the software (i.e., libraries and programs) as it was deployed on these systems. Table 2 lists all security-critical system calls with the integrity checking level that we used during the evaluation. For some less critical and frequently invoked system calls, we chose to only check for code integrity.

Checking Level	System Calls
Code integrity	creat, write(v), fork, sendfile, unlink, open, send, sendmsg, sendto
Code + Symbol integrity	execve, mmap, mprotect, ioctl, connect, socket, delete_module, init_module, symlink, chmod, chown, kill, reboot, accept, dup, pipe, socketpair, socketcall, ipc

Table 2: Integrity checking levels used for the evaluation of BINtegrity.

We first provide an insight into the cost of BINtegrity's internal operations before we evaluate BINtegrity's impact on these systems using benchmarks.

Internal Operation The evaluation of BINtegrity's internal operation, specifically the invariant extraction, is based on both micro-benchmarks and real-world scenarios. For the micro-benchmark, we used lmbench's [12] system call latency measurement on the WZR-HP-G450H for the `write` system call. For the real-world scenarios, we picked two typical applications, a web server and a web browser, each running on a different system that is protected by BINtegrity. We had BINtegrity collect statistical data for each process running in the system. During a two-minute evaluation time period, the web server received two requests on a page, and the web browser loaded a single page to simulate a typical usage scenario.

Invariant Extraction and Caching Invariant extraction is time consuming. It requires costly operations such as file parsing as well as disassembling and emulating code. To alleviate its impact on BINtegrity's performance, we attempted to reduce invariant extraction to a necessary minimum. On one hand, we allow fine-tuning of BINtegrity to a specific platform's performance requirements by setting the checking levels. On the other hand, invariant extraction is only performed once per program state by caching its result. The drastic effect of caching is shown in Figure 5. As can be seen, checks based on the results of the code invariants as well as both code and symbol invariants only add little additional overhead once they are cached.

Frequency and Distribution of Invariant Extraction As mentioned before, the number of invariant extractions during program execution is critical to BINtegrity's performance. Table 3 shows how often code (for L2 checks) as well as code and symbol invariant extractions (for L3 checks) were performed. In spite of the complexity of the applications tested and the usage scenario, the numbers are quite low. In combination with caching, BINtegrity only has to carry out fewer than 100 performance-critical invariant extractions in each case.

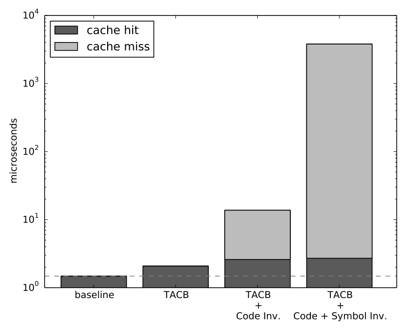


Fig. 5: Effect of invariant caching measured in system call latency. Once cached, checking invariants only incurs marginal overhead.

Invariant Extractions	Code	Code + Symbol	Args
Android Browser	21	53	0
Nginx Web Server	19	24	10

Table 3: Number of invariant extractions executed for two typical applications. Args refers to the number of static argument assignments that were enforced.

Figure 6 shows the distribution of invariant extractions over time. We can see that a large number of extractions are executed at process startup, with the next spike occurring when the application first executes its main functionality. After a few seconds, the main code paths have been executed at least once. Hence, they do not require any further extractions during normal operation and, thus, unnecessary overhead is avoided.

Memory Overhead For each process in the system, BINtegrity reserves space to cache invariants for up to 257 code points. Every code point requires 40 bytes of storage for statistics that we use to measure our runtime performance. The memory requirement for this cache is 10 KB per process. During our evaluation, we never encountered more than 74 code points being cached for a process. The unused cache space provides enough storage to handle applications that require caching invariants for more code points while not wasting resources as the the overall memory usage is relatively small. We consider tuning the size of the cache as part of specializing BINtegrity for a specific platform. We further require a small amount of memory to store our per process TACB. The memory requirement for each TACB entry is 16 bytes. During our evaluation, we never encountered a process with more than 100 entries stored in its TACB, thus staying below 2 KB per process. In total, BINtegrity adds a memory overhead of around 12 KB for each process. This memory overhead is negligible if compared with the memory required through the use of shared libraries, some of which easily occupy a few hundred kilobytes.

Benchmarks To measure BINtegrity's impact on system performance, we ran various benchmarks covering both specific metrics (e.g., disk I/O) and overall performance.

Disk I/O For the disk I/O performance evaluation, we used Bonnie++, configured to use a filesize of 500 MB for access, and 16,348 files for creation/deletion benchmarks. Being platform independent, we ran it on both ARM and MIPS. For the ARM implementation, the worst-case overhead is 10% for random seeks. On MIPS, the worst-case overhead is 20% for block write operations.

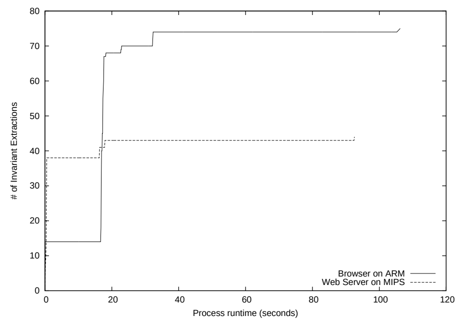


Fig. 6: Invariant extractions over the lifetime of two typical applications. For both applications, we show that extractions are executed at program startup and at the point where the application performs typical activity for the first time. After a code path has been executed once it causes no further extractions.

Network I/O For the WZR-HP-G450H, we used the Apache benchmark from a separate computer on the LAN to request a 128 KB document. The document was served by the default *nginx* installation running on the router. The average overhead measured over 1000 requests was 2.03%.

Whole-System Performance On the Galaxy Nexus Android phone, we used the popular AnTuTu benchmark[6] to measure performance overhead introduced by BINtegrity. The AnTuTu benchmark measures a variety of system components such as the Android runtime and the I/O subsystem. The benchmark result shows that BINtegrity only incurs an overhead of 1.2% compared to the baseline.

8 Discussion and Limitations

In general, BINtegrity does not prevent code execution from arbitrary memory locations, but restricts the invocation of kernel services. Hence, BINtegrity mitigates attacks that interact with the operating system, but does not prevent attacks against the application logic itself. In the following, we discuss both how BINtegrity deals with certain code constructs and which aspects could hamper its effectiveness.

Call Stack Depth The call stack BINtegrity relies on to extract data from the binary is only two levels deep. Undoubtedly, a deeper call stack would enable us to perform more thorough integrity checks. However, a deeper call stack would require keeping track of return addresses in the process' memory, which both contradicts our threat model and in addition slows down analysis.

A study of two lightweight C library implementations that are popular in the embedded world, uClibc and bionic, shows that only a relatively small subset of the library functions use indirection, i.e. call another function before invoking the system call (Figure 4). Besides, these indirections can be removed entirely by recompiling the C library with inlining.

Forward Emulation BINtegrity's effectiveness is determined by the information provided by the forward emulation. Some system call wrappers will degrade information on system call arguments by performing operations that our register-based code emulation cannot track. While for uClibc, the number of wrappers that modify might seem high,

[6] AnTuTu `https://play.google.com/store/apps/details?id=com.antutu.ABenchMark`

we note that only five out of them are security-critical. We plan to address this issue in future work by enhancing our static analysis with a lightweight memory model.

Forging the Link Address An attacker who is aware of BINtegrity could try to set ret_{lr} to another value. However, only valid values would pass all steps of the invariant enforcement, i.e. stemming from a correct origin, adhering to call chain integrity and argument integrity and be imported based on the rules of symbol integrity. If the attacker succeeds in finding such an address, she would lose control flow control after the function returns and thus be limited to a single function invocation.

System call wrappers	Bionic	uClibc
Total	194	243
Using indirections	71	31
Modifying arguments	1	69

Table 4: Characteristics of C library system call wrappers that degrade the detail level of the extracted invariants. The numbers were derived from the binaries we found on Android (Bionic) and OpenWRT (uClibc). Indirections can be reduced to zero by recompiling the library.

Just-In-Time Compilation Just-In-Time (JIT) compilation, best known for speeding up Javascript and ActionScript, is rarely found on embedded devices. Although, in theory, JIT compiled code directly violates BINtegrity's fundamental idea of only executing code that is present in the binary on disk, BINtegrity does not break JIT. The reason is that JIT compiled code never directly interacts with the standard C library. At the same time, BINtegrity prevents JIT-ed code from direct interaction with the standard C library or the system call interface

9 Conclusions

In this paper, we presented BINtegrity, a novel approach to exploit mitigation that is specifically tailored towards embedded systems that are based on common RISC architectures. BINtegrity works by extracting a combination of static and dynamic properties relevant to OS service requests from executables and enforcing them during runtime.

We leverage common properties of the RISC architecture to design and build an exploit mitigation system that is practical and low-overhead and thus lends itself specifically for the use in systems with limited resources.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-Flow Integrity. In: ACM Conference on Computer and Communications Security (CCS) (2005)
2. Andersen, S., Abella, V.: Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies (2004)
3. Baume, T.: Netcomm NB5 Botnet Psyb0t. `http://baume.id.au/psyb0t/PSYB0T.pdf`
4. Cheng, Y., Zhou, Z., Yu, M., Ding, X., Deng, R.: ROPecker: A Generic and Practical Approach for Defending against ROP Attacks. In: Network and Distributed System Security Symposium (NDSS) (2013)
5. Craig Heffner: OSVDB 86824 Exploit. `http://www.devttys0.com/wp-content/uploads/2012/10/dir605l_exploit.txt`
6. Davi, L., Hanreich, M., Paul, D., Sadeghi, A.R., Koeberl, P., Sullivan, D., Arias, O., Jin, Y.: HAFIX: Hardware-assisted flow integrity extension. In: Proceedings of the Annual Design Automation Conference (2015)

7. Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W., Gong, W.: Anomaly Detection Using Call Stack Information. In: IEEE Symposium on Security and Privacy (Oakland) (2003)
8. Goektas, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out Of Control: Overcoming Control-Flow Integrity. In: IEEE Symposium on Security and Privacy (Oakland) (2014)
9. Jacob Holcomb: CVE-2013-465 Exploit. `http://www.exploit-db.com/exploits/27133/`
10. Kruegel, C., Mutz, D., Valeur, F., Vigna, G.: On the Detection of Anomalous System Call Arguments. In: European Symposium on Research in Computer Security (ESORICS) (2003)
11. Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-Pointer Integrity. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2014)
12. McVoy, L., Staelin, C.: Lmbench: Portable tools for performance analysis. In: USENIX Annual Technical Conference (USENIX ATC) (1996)
13. Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent ROP exploit mitigation using indirect branch tracing. In: USENIX Security Symposium (USENIX SEC) (2013)
14. Provos, N.: Improving host security with system call policies. In: USENIX Security Symposium (USENIX SEC) (2003)
15. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.R., Holz, T.: Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In: IEEE Symposium on Security and Privacy (Oakland) (2015)
16. Shu, X., Yao, D., Ramakrishnan, N.: Unearthing stealthy program attacks buried in extremely long execution paths. In: ACM SIGSAC Conference on Computer and Communications Security (CCS) (2015)
17. Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: Eternal War in Memory. In: IEEE Symposium on Security and Privacy (Oakland) (2013)
18. Team Cymru: SOHO Pharming. `https://www.team-cymru.com/ReadingRoom/Whitepapers/2013/TeamCymruSOHOPharming.pdf` (2014)
19. Ullrich, J.: Linksys Worm The Moon. `https://isc.sans.edu/forums/diary/Linksys+Worm+TheMoon+Summary+What+we+know+so+far/17633` (2014)
20. Vaughan, J.A., Hilton, A.D.: Paladin: Helping Programs Help Themselves with Internal System Call Interposition (2010)
21. van der Veen, V., Andriesse, D., Göktaş, E., Gras, B., Sambuc, L., Slowinska, A., Bos, H., Giuffrida, C.: Practical Context-Sensitive CFI. In: ACM Conference on Computer and Communications Security (CCS) (2015)
22. van der Veen, V., dutt Sharma, N., Cavallaro, L., Bos, H.: Memory Errors: The Past, the Present, and the Future. In: Conference on Research in Attacks, Intrusions, and Defenses (RAID) (2012)
23. Wagner, D., Dean, D.: Intrusion Detection via Static Analysis. In: IEEE Symposium on Security and Privacy (Oakland) (2001)
24. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: ACM Conference on Computer and Communications Security (CCS) (2002)
25. Zhang, M., Sekar, R.: Control flow integrity for COTS binaries. In: USENIX Security Symposium (USENIX SEC) (2013)
26. Zhou, Y., Wang, X., Chen, Y., Wang, Z.: ARMlock: Hardware-based Fault Isolation for ARM. In: ACM Conference on Computer and Communications Security (CCS) (November 2014)