2015 IEEE Symposium on Security and Privacy

High System-Code Security with Low Overhead

Jonas Wagner*, Volodymyr Kuznetsov*, George Candea*, and Johannes Kinder†
*School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne
†Department of Computer Science
Royal Holloway, University of London

***Abstract*—Security vulnerabilities plague modern systems because writing secure systems code is hard. Promising approaches can retrofit security automatically via runtime checks that implement the desired security policy; these checks guard critical operations, like memory accesses. Alas, the induced slowdown usually exceeds by a wide margin what system users are willing to tolerate in production, so these tools are hardly ever used. As a result, the insecurity of real-world systems persists.**

We present an approach in which developers/operators can specify what level of overhead they find acceptable for a given workload (e.g., 5%); our proposed tool ASAP then automatically instruments the program to maximize its security while staying within the specified "overhead budget." Two insights make this approach effective: most overhead in existing tools is due to only a few "hot" checks, whereas the checks most useful to security are typically "cold" and cheap.

We evaluate ASAP on programs from the Phoronix and SPEC benchmark suites. It can precisely select the best points in the security-performance spectrum. Moreover, we analyzed existing bugs and security vulnerabilities in RIPE, OpenSSL, and the Python interpreter, and found that the protection level offered by the ASAP approach is sufficient to protect against all of them.

I. Introduction

System builders routinely need to satisfy conflicting demands of performance, productivity, and security. A lot of systems code is written in unsafe languages, like C/C++, because they have low runtime overhead, they enable low-level access to hardware, and because they are sometimes the only way to use legacy libraries or tool chains. The drawback is that unsafe languages burden the programmer with managing memory and avoiding the many behaviors left undefined by the language specifications. This makes it especially hard to write secure software; but the security of the entire software stack depends on input parsers, language runtimes, cryptographic routines, web browsers, OS kernels, and hypervisors written in these languages. The quest for performance and low memory consumption can often compromise safety, and even extensive test suites and the use of tools like Valgrind still leave holes in the code. It is thus not surprising that buffer overflows are still the #1 vulnerability exploited by attackers [26] and that new ones have been revealed to take control of browsers and OSs in every edition of the Pwn2Own contest [28] since 2007.

Developers do have the option to employ techniques for "retrofitting" security and safety into their software. Tools like AddressSanitizer [32], StackGuard [9], SoftBound [25], WIT [3], SafeCode [10], UndefinedBehaviorSanitizer [8], Code-Pointer Integrity [20] etc. insert *sanity checks* into the code to verify at run-time that desired safety properties hold. These checks might verify that array indices are in bounds, that arithmetic operations do not overflow, or that data structure invariants hold. If a sanity check fails, it typically is unrecoverable, and the program is aborted. Other than that, sanity checks do not affect the program state.

Unfortunately, such approaches are hardly ever used in production because of their overhead. The introduced sanity checks slow down the program and completely erase the performance gains that come from low-level languages. Programmers today are faced with a binary choice: fast and insecure, or slow and safe.

This is a pity, because program instrumentation can often be made *elastic.* Instrumentation tools introduce many small, independent checks. By carefully selecting which checks to use, developers could control the overhead and trade some security to satisfy their performance constraints. Except that developers lack a principled way to reason about the impact of checks and choose the most effective ones.

We introduce ASAP, the first fully-automated approach for instrumenting programs subject to performance constraints. It allows developers to specify an *overhead budget*, and then automatically profiles and selects checks such as to build a program that is *as secure as possible* for the given budget. With ASAP, developers can precisely choose the optimal point in the security-performance trade-off.

It is often possible to obtain high security at low overhead, for two reasons: First, the checks that are most important for security are checks guarding obscure, untested, buggy code where protection is most needed. Because this code is typically cold, the checks are rarely executed and do not contribute much to the overhead. Second, most of the induced overhead comes from only few expensive checks located inside hot loops. These checks are executed over and over again, burning cycles while likely not adding much to the program's security.

ASAP allocates the fixed overhead budget to checks in cold parts of the program to maximize security. We found that this approach works particularly well for CPU-intensive tasks such as parsing input, encoding or decoding data, or performing cryptography. In these cases, ASAP can select 87% of the available sanity checks on average, while keeping the aggregate overhead below 5%, which is an order of magnitude lower than existing solutions. Because these tasks often process untrusted data, we believe that the ASAP approach enables real security benefits in today's production environments.

DOI 10.1109/SP.2015.58

IEEE computer society

ASAP quantifies the resulting security by computing the *sanity level.* This is the fraction of potentially vulnerable program instructions (e.g., memory accesses) that is protected by a sanity check. Our experiments provide evidence that the sanity level is a lower bound on the fraction of vulnerabilities or bugs that will be detected by sanity checks. This lower bound holds because bug density is higher in cold code than in hot code, as substantiated by our study of bugs in the Python interpreter and by security vulnerabilities from the CVE database (§VI).

We built a prototype of the ASAP approach based on the LLVM compiler framework [21]. It supports checks inserted by AddressSanitizer, UndefinedBehaviorSanitizer, and SoftBound. Like these tools, ASAP works at the LLVM intermediate representation level. We have tested it on a number of C/C++ programs such as the Python interpreter, OpenSSL, Apache, NGINX, and the Phoronix and SPEC benchmarks. ASAP uses a profiling workload to measure which checks are most expensive. For best performance, users should choose a profiling workload that is close enough to the production workload to identify all expensive checks; we found that using a program's test suite often works well. The process of compiling, profiling, and check selection is fully automated.

Not all programs and instrumentation tools work well with ASAP. In some cases, a large fraction of the overhead is not due to security checks themselves, but comes from other sources like changes to the memory allocator, increased thread startup time, or maintaining metadata required by checks. We call this overhead *residual overhead*. We discuss its causes in §III, and also give design principles to make future instrumentation tools more elastic.

ASAP can be used today to increase the security of software we run in production. It identifies cases where a surprising amount of security can be gained for a very low price. We will make the ASAP source code publicly available. We hope this leads to a world without security vulnerabilities like Heartbleed: with ASAP, the Heartbleed vulnerability would have been avoided with only 5% reduction in web server throughput.

This paper makes the following main contributions:

- We show that program instrumentation can be made elastic, so that users can choose how much to pay for the security they need. Our tool ASAP is a practical, automated way to navigate the security vs. performance trade-off, reducing the entry barrier for applying instrumentation-based security mechanisms to systems code.
- We study existing bugs and security vulnerabilities (in Python and CVEs for open source software) and show that about 95% lie in cold code, where protection is cheap.
- We show that, in practice, a protection level comparable to that of the strongest tools for retrofitting language security can be achieved at a fraction of the overhead.

The rest of the paper provides background information and discusses related work (§II), describes the design of ASAP (§IV) and our ASAP prototype (§V), evaluates ASAP on several benchmarks (§VI), discusses multiple extensions (§VII), and concludes (§VIII).

II. Background and Related Work

There are many aspects to the security of software systems and many ways to improve it. We focus on *sanity checks*, which verify safety properties at runtime. These properties may relate to undefined behavior in low-level languages or just generally to invalid states in the application logic. The distinguishing factor of a sanity check is that once it fails, the only safe option is to abort the program, because continuing the execution would likely lead to dangerous undefined behavior. We give a formal description of sanity checks in §IV-C. Sanity checks are used in various forms; in the following, we present prominent examples of sanity checks, several of which address shortcomings of the security of low-level code by guarding various types of critical operations.

Data Execution Prevention. A widely deployed sanity check supported by hardware is Data Execution Prevention (DEP) [22]. In a DEP-protected system, a process's data pages are marked non-executable in the system page table. The CPU raises a hardware exception if an instruction from such a page is about to be executed. This thwarts attacks that write attacker-supplied code (so-called shellcode) to the process's memory and redirect execution to this code. DEP has to manage the executable flag, which is essentially a form of annotation or metadata, and it requires a (hardware) check before each critical operation, i.e., before executing an instruction. For less than 1% overhead, DEP protects against basic forms of remote code execution attacks, but it can be circumvented relatively easily using more elaborate attacks based on "return-oriented programming" [35].

Assertions. Developer-provided assertions are the most common type of sanity check in code. C. A. R. Hoare reported in 2002 that over 250,000 assertions are present in Microsoft Office [13]. Assertions incur runtime overhead, so they are often disabled in production. To meet the conflicting demands of safety and performance, some projects have found manual ways to enable only "cheap" assertions. For example, the LLVM project builds with assertions by default, but has an `XDEBUG` preprocessor flag to enable additional, more expensive assertions.

Many safety properties are better enforced *mechanically* by tools, instead of manually by the programmer. Tools have the advantage that they can guard *all* critical operations with sanity checks; for instance, a tool can automatically insert a check before every arithmetic operation to verify that there is no overflow. This rules out entire categories of bugs and vulnerabilities. Other than the fact that they are inserted automatically and not visible in source code, the structure and effect of such checks is similar to assertions.

Undefined Behavior Checks. UndefinedBehaviorSanitizer [8] (UBSan) instruments a program with checks that ensure the absence of various operations that are undefined in the C/C++ language standards, and thus generally unsafe to use. UBSan catches NULL pointer dereferences, unaligned

Safety	Tool	Overhead	ASAP
Low	DEP	<1%	
	Stack Canaries	<1%	
High	WIT	7%	
	CPI	8%	
	SAFECode	10%	
	ASan	73%	✓
	UBSan	71%	✓
Full	SoftBound/CETS	116%	✓

Fig. 1. Automatic solutions to enforce program safety, classified according to the strength of their safety guarantees. A check mark in the last column indicates that our current ASAP prototype includes support for the tool.

memory accesses, signed integer overflows, out of bound bit shifts, etc. These problems are less frequently exploited by attackers than memory errors. Yet they can lead to security vulnerabilities, e.g., when an integer overflow occurs while computing the size of an object, and thus the wrong amount of memory is allocated. Wang et al. [36] found that compilers can transform undefined behavior into security vulnerabilities by "optimizing away" security-critical code. Checks inserted by UBSan are stateless and do not require metadata. We measured their overhead to be 71% on average on the SPEC CPU 2006 benchmarks.

Stack Canaries. Another widely used application security mechanism are stack canaries [9] and the related Structured Exception Handler Override Protection (SEHOP) [33]. Stack canaries detect when function return addresses have been overwritten. Compiler-inserted code in the function prologue inserts a secret value just before the function return address on the program stack. Before returning, the function verifies that the secret value is still in place. Buffer overflows overwriting the return address will very likely modify the secret value and are thus detected before the attacker gains control.

Stack canaries manage metadata in the form of loading the secret value onto the stack, and they require an extra check before a critical operation, the function return. Stack canaries incur below 1% overhead. They offer some protection against simple buffer overflows, but they can be neutralized by modifying attacks, e.g., by directly overriding the return address [35].

Memory Safety Checks. Stronger forms of defense retrofit memory safety to C and C++ code by protecting all memory accesses with sanity checks. The available tools instrument a target program such that they insert code before each memory access to check whether the address is valid. The strictness of this definition of "valid" influences the provided safety guarantees and the performance overhead.

Some of the earliest examples of such tools are BCC [19], `rtcc` [34], SafeC [6], and the Jones and Kelly bounds checker [17]. CCured [27] is one of the first systems to reduce overhead by avoiding dynamic checks. It attempts to statically type check pointer accesses in a C program to prove that they are safe, and only inserts dynamic sanity checks where it cannot prove safety. It commonly requires adjustments in the target program, but it provides a formal guarantee of memory safety. Cyclone [16] and later Rust [30] continue along this path. They can remove even more runtime checks in a sound way by providing safe language features.

SoftBound CETS [24], [25] provides the same guarantee but is designed for compatibility and to not require adjustments in the target program. SoftBound associates bounds with every pointer in the program, i.e., it keeps track of which memory region the program may legally access through a particular pointer. It inserts code to maintain metadata whenever a pointer is created or copied. Additionally, SoftBound uses metadata to guarantee that the object being accessed has not been freed in the meantime. In exchange for its comprehensive guarantees, SoftBound has the highest overhead of all tools described here. The authors report 116% on average.

Strong guarantees come with high overhead, thus other approaches achieve lower overhead by weakening the guarantees provided: Write Integrity Testing (WIT) [3] restricts the possible target addresses of memory stores to a set of valid objects that are statically determined at compile time. The limitation to stores allows to reduce the overhead to 7% on average; however, exploits of pure information leakage vulnerabilities would remain undetected. In a similar spirit to WIT, SAFECode [10] enforces statically computed aliasing relationships, and also reports overheads below 10%. CRED [31] restricts its sanity checks to string accesses only.

AddressSanitizer (ASan) [32] does not enforce full memory safety, but prevents memory accesses from overflowing into adjacent memory areas. ASan inserts forbidden areas (so-called red zones) between objects in memory to detect buffer overflows. Before each memory load or store, ASan consults its shadow memory (a compact lookup table storing whether an address is red or not) to ensure the program does not access any red zones. Additionally, recently `free`'d areas are marked red, so that use-after-free bugs can be detected. Maintaining the red zones and the shadow memory, changing the memory allocator, and performing access checks causes an average slowdown of 73%.

Baggy Bounds Checking [4] achieves efficient checks by padding all memory object sizes to powers of 2. Its sanity checks do prevent an overflow from affecting other memory objects, but a vulnerability or attack may go undetected if the overflow stays within the padding.

Code-pointer Integrity [20] is a protection mechanism that enforces partial memory safety. It protects just enough memory areas to prevent attackers from overriding a return address or pointer to function. This thwarts control-flow hijack attacks at 8.4% overhead. It does not prevent other values in memory to be overridden; for example, the recent GHOST exploit for the Exim mail server [2] would still succeed, because it overrides an access-control list instead of a code pointer.

Finally, Control Flow Integrity [1] forgoes memory safety altogether and only forces control flow transfers to remain within a known set of safe target locations. It therefore only prevents attacks that attempt to divert control flow, e.g., to some shellcode or exploit gadgets.

ASAP can be used with any tool that inserts sanity checks into programs; its purpose is to elide the checks that provide the lowest safety return on induced overhead, until the estimated overhead fits within the given overhead budget. In our evaluation in §VI, we show the effectiveness of our ASAP prototype on ASan and UBSan.

The Cold Code Hypothesis. A number of tools are, like ASAP, based on the assumption that bugs are more frequent in cold code. This is confirmed by studies such as [38], which found that 30% of the analyzed catastrophic failures were caused by wrong error-handling code.

Developers can use methods such as bias-free sampling [18] or adaptive statistical profiling [7] to focus debugging efforts and program analysis on cold code. Similarly to ASAP, the Multicompiler project [14] improves a program's security by focusing efforts on the cold parts of the program.

III. The Sanity/Performance Trade-Off

Several of the tools for retrofitting security discussed in §II trade off security against performance. Policies like WIT or CFI settle for enforcing a weaker security guarantee for the entire program to reduce overhead.

ASAP takes a different approach and treats the checks inserted by an underlying strict tool uniformly. Based on profiling information, ASAP removes the most expensive sanity checks, but leaves all others unchanged. Thus, the remaining checks still have the chance of preventing exploits that would have been possible against systems that globally enforce a weaker policy. The downside is that ASAP cannot provide a formal security guarantee. However, we argue that using ASAP can make a program safer than using a tool that achieves low overhead by globally providing a weaker level of protection. In §VI-D1, we provide empirical evidence by showing that a version of OpenSSL hardened by ASAP detects the Heartbleed vulnerability at only 5% overhead. Weaker policies such as WIT or CFI could not have prevented the vulnerability.

Understanding the practical security achieved by a form of instrumentation is difficult. When we apply an instrumentation that provides a formal guarantee, such as full spatial memory safety, we can say that we have ruled out a particular class of attacks. However, the instrumented program is by no means guaranteed to be perfectly secure. Other classes of attacks may very well still be possible. We cannot clearly assign a number to the security provided by any such instrumentation. In this light, the level of security provided by ASAP should be seen as orthogonal to the classes of formal guarantees enforced by typical instrumentations. Instead of trading off performance against classes of protection, it trades off performance against individual sanity checks. Whether one or the other prevents more damage to a system depends on the number and type of vulnerabilities they prevent after being deployed. Therefore we argue that ultimately the practical security afforded by an instrumentation has to be evaluated empirically, which we do in §VI-D.

Reasoning about the trade-off between sanity and performance that ASAP provides requires that we quantify the contributions of sanity checks to security and to performance overhead. We would like a metric that informs us just how much performance improves and how much safety decreases when removing a particular check.

The impact of a single sanity check can vary significantly; for instance, a single assertion in the Firefox browser caught as many as 66 bugs [29]. Sometimes multiple assertions would prevent the same vulnerability; for example, an exploit for the recent CVE-2014-2851 vulnerability in the Linux kernel first causes a reference counter to overflow, which later leads to a use-after-free bug. Different assertions detect each of these problems, and the exploit only succeeds if both assertions are absent.

In principle, the contribution of a sanity check to safety is its potential for detecting safety violations. Hence, the only valuable sanity checks are those that guard potentially vulnerable operations that could compromise security. Without having further information on the likelihood of operations to be vulnerable, we consider all sanity checks of critical (i.e., potentially vulnerable) operations like memory accesses to be of equal value. We thus define the *sanity level* of a program as the fraction of its critical operations that are guarded by sanity checks. For a given tool that instruments all critical operations, the sanity level is thus the fraction of (static) checks that are still present.

Note that this metric makes no attempt to represent actual failure probabilities. Rather, the sanity level makes a statement about the static protection level of a program similarly to how statement coverage makes a statement about the quality of a test suite. ASAP considers only the estimated run-time cost when choosing which checks to eliminate, so the accuracy of the sanity metric does not affect the resulting programs. We use the sanity level only as an easily measurable indication of protection remaining in the program at a given overhead. A more reliable assessment of the effectiveness of the protection can only be made empirically using real vulnerabilities, as discussed above.

The choice of providing no security guarantees liberates ASAP from constraints that make existing solutions too slow to be widely adopted in practice. It enables users to weigh security benefits against performance.

We quantify the performance impact of a given sanity check by estimating its run-time cost in terms of CPU cycles. However, a sanity check can also impact performance in other ways: (1) checks depend on metadata that needs to be computed and propagated; (2) the metadata needed by checks occupies registers and cache lines, leading to higher register pressure and more cache misses; (3) instrumentation tools incur some fixed overhead. For example, every time a program spawns a new thread, AddressSanitizer needs to set up metadata for the thread's stack; (4) instrumentation tools may modify memory allocators, function calling conventions, the runtime library, or the program's memory layout. Each of these modifications can affect performance.

We estimate run-time cost via CPU cycles only for two reasons. First, the CPU cycles required to execute individual checks allow to estimate their cost relative to the total instrumentation cost, which is all that is needed for ASAP. Second, ASAP does not yet affect metrics like memory overhead, and so it has not yet been necessary to measure them.

ASAP's goal is to allow system developers fine-grained control over the sanity/performance trade-off. Each of the tools we analyzed provides manual ways to tune performance. ASan, UBSan and SoftBound allow individual functions to be excluded from instrumentation. Users can tweak the size of red zones in ASan, disable a specific type of check in UBSan, or instrument only write operations in SoftBound. Each of these tweaks are coarse-grained in the sense that users cannot target individual checks. They are also imprecise because users cannot measure the impact of individual checks on performance. We show in §VI that ASAP's automated, profiling-based approach only needs to eliminate a small fraction of checks to significantly reduce overhead; e.g., removing the top 5% most expensive checks reduces overhead by 76% on average for the benchmarks in our evaluation.

Our analysis of 2014's CVE entries (§VI-F) and of several security bugs in Python provides empirical evidence that most bugs lurk in cold regions of the program; the sanity checks that prevent them from being exploited thus often cause only little run-time overhead.

IV. Design

ASAP takes as input a software system and a workload, as well as one or several instrumentation tools. It then applies these tools to obtain a full set of available sanity checks. After estimating the cost of checks by profiling, ASAP then selects and applies a maximal subset such that the combined overhead is within budget. The remainder of this section presents the ASAP workflow in detail (§IV-A), discusses the choice of workload for profiling (§IV-B), introduces the concept of sanity checks (§IV-C), and explains how ASAP's design choices affect its effectiveness (§IV-D).

A. The ASAP *Workflow*

A user of ASAP starts with a software system that is to be protected using one or several instrumentation tools. We designed ASAP to be part of the software's compilation process, just like the instrumentation tools described in §II. Compilation using ASAP consists of three steps: *instrumentation, profiling,* and *check selection*. The initial steps are illustrated in Figure 2.

1) Instrumentation: The user starts by compiling the target program with full instrumentation enabled. This step depends on the specific instrumentation tool, but can be as simple as adding an additional compilation flag (for ASan, SoftBound, and UBSan). This leads to a binary (or several) that is protected, but too slow to run in production. ASAP automatically recognizes the sanity checks in the program in order to measure and process them further.

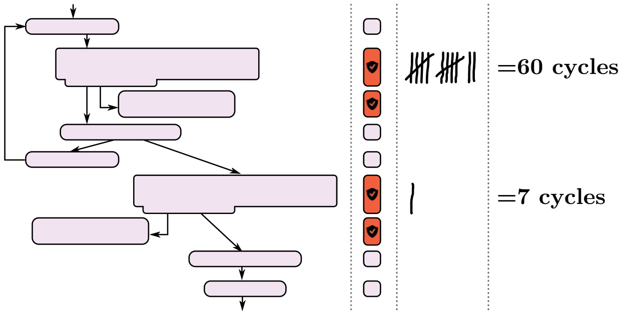


Fig. 2. Recognizing sanity checks and measuring their cost. The figure show an example control-flow graph fragment of an instrumented program. ASAP first recognizes all the sanity checks (shown in red) by their structure. During profiling, ASAP counts how often each instruction in these checks is executed. It then uses these counts to estimate the amount of time spent due to each check.

2) Profiling: The second step consists of profiling the application against a suitable workload and computing the cost of each check. To obtain profiling data, ASAP further instruments the program from step 1 with profiling counters. Similar to GCOV [11], it adds one counter per edge between basic blocks. Before each branch, ASAP inserts an increment of the corresponding counter.

Once the profiling run finishes, ASAP computes from the counter values the number of times any given instruction in a sanity check has been executed. By multiplying this value with a static estimate of the CPU cycles required to execute that instruction, it computes the accumulated cost for that instruction. The total cost in CPU cycles of a given sanity check is then the sum of the costs of the instructions inserted by that check. The sum of the costs of all sanity checks in the program gives the total number of cycles spent in checks while executing the profiling workload with the fully instrumented program.

To accurately estimate the percentage of overhead due to each check, ASAP first measures the maximum overhead o_{max} at full instrumentation. The maximum overhead results from measuring the running time of the software with full instrumentation (but no profiling counters) and subtracting its running time when executed without any instrumentation at all. Many instrumentation tools also have a fixed overhead o_{min} that is independent of the checks (e.g., for metadata management). ASAP measures this minimum overhead by running a version of the software that is instrumented but had all actual checks removed.

ASAP uses the data from these three profiling runs to determine the fraction of the total cycles spent in checks that can be preserved without exceeding the overhead budget, which we call the *cost level* c. The overhead o is a linear function of c:

$$o = o_{min} + c \cdot (o_{max} - o_{min})$$

Our experimental results confirm that this linear relationship holds and thus the cycle-based estimation of check cost is

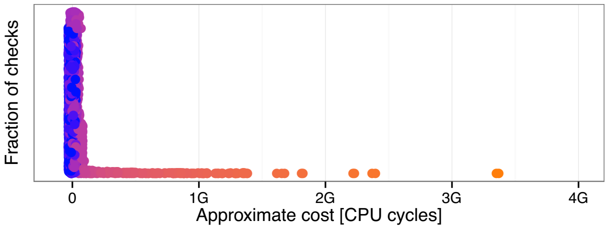


Fig. 3. Dotplot of the cost of the 3864 sanity checks in the bzip2 benchmark. A dot's position on the x axis corresponds to the corresponding check's cost in 10^9 cycles. The single most expensive check is as expensive as the 3414 cheapest checks together.

precise enough to give ASAP accurate control of the incurred overhead. For a given target overhead o, the target cost level c can be computed by transforming the above equation:

$$c = \frac{o - o_{min}}{o_{max} - o_{min}}$$

3) Check Selection: Knowing the cost of each check and the target cost level, ASAP now uses a simple greedy algorithm to compute a maximal set of checks to preserve, while staying within the overhead budget. It orders checks by cost and preserves them starting with the cheapest check, as long as the fraction of the total check cost allowed by the cost level c is not exceeded. Because the distribution of check cost is highly skewed, it is possible to preserve a fraction of checks that is much larger than the fraction c of the total cost. Figure 3 shows a typical distribution of check cost in a program, where a few hot checks dominate the cost of the instrumentation.

ASAP eliminates all checks that have not been preserved by removing them from the instrumented program generated in step 1. It then re-optimizes the program using standard compiler optimizations. This ensures that all data computed solely for use by those sanity checks is also removed from the program. The result is an optimized, production-ready executable.

When production workloads have significantly changed from what was used during profiling, steps 2 and 3 can be repeated with an updated workload to re-estimate the performance trade-off and produce a newly adapted binary.

B. Workload Selection

Profiling identifies a set of hot regions in a program. For optimal results, the checks in these regions should be the ones that are most expensive in production, and the ones that contribute least to security. These two requirements are often well aligned in practice, and can be used as guidelines to select an ideal profiling workload.

ASAP is based on the assumption that a few sanity checks are much more expensive than others. For ASAP to meet tight overhead budgets, the profiling workload must expose the expensive checks. This means that it should execute all performance-critical program components. The specific way in which they are executed does not matter, because ASAP only depends on expensive checks being sufficiently visible, not on precisely measuring their runtime. Naturally, a profiling workload representative of real workloads will yield the best results in production.

Our confidence in the security provided by the remaining checks is based on the assumption that checks in cold code are more likely to catch bugs than checks in hot code. For this to hold, it is important that the program parts stressed by the profiling workload are also well tested. This is often implicit, but can be made explicit by using the program's test suite as the profiling workload. The test suite does not need to have high coverage, as ASAP will preserve checks in uncovered parts. However, it should provide assurance that the covered parts are indeed correct, e.g., by using a large number of assertions to audit the program state.

Developers can use these ideas to actively guide ASAP. For example, by increasing test coverage for performance-critical program parts, confidence in the correctness of these parts increases and the need for safety checks decreases. ASAP will exploit this and assign checks to other areas where they are needed more, thereby improving performance.

C. Sanity Checks

To understand how ASAP works and what it assumes, we define a sanity check to be a piece of code that tests a safety condition and has two properties: (1) a passing check is *free of side-effects*, and (2) a failing check *aborts* the program. This characterization of sanity checks has important implications: First, ASAP can automatically recognize sanity checks in compiled code. Second, removing a sanity check is guaranteed to preserve the behavior of the program, unless the check would have failed. Note that metadata updates are not part of sanity checks by this definition (and can thus remain as residual overhead).

The sanity checks seen by ASAP do not necessarily correspond exactly to operations in the program source, since it runs after compiler optimizations have been already applied. ASAP benefits from its tight integration with the compiler. Depending on their type, the compiler may be able to eliminate certain sanity checks on its own when they are impossible to fail. Other transformations such as function inlining can duplicate static sanity checks in the compiled program. This refines the granularity of ASAP: if there are multiple copies of the same function, ASAP can distinguish the individual call sites and may choose to optimize only some of them.

D. Design Choices

ASAP is *tool-agnostic* and can work with all mechanisms that insert sanity checks into the program. We tested our prototype using AddressSanitizer, SoftBound and UndefinedBehaviorSanitizer, and verified that the checks inserted by WIT and SafeCode (see §II) also follow the same structure.

We designed ASAP to be a *compiler-based solution*. The advantage of source code access over a pure binary solution is that ASAP can thoroughly re-optimize programs once expensive checks have been removed. This allows additional dead code elimination, reduces register pressure, and can make hot functions small enough to be inlined. In principle, however, ASAP's approach would also work with a powerful binary rewriting and reoptimization system in the spirit of SecondWrite [5].

ASAP relies on *profiling*. We chose this approach because it is a reliable way to obtain the cost of a check, and makes no assumptions about the nature of the program or the structure of sanity checks. However, it requires an adequate workload and increases build times. For many projects, a workload is available in form of a test suite; this special case has interesting implications for security and is discussed below.

We ensured ASAP is *practical*. It can be applied to any system for which the underlying instrumentation tool works, and it does not add any restrictions of its own. Safety checks may soon make use of upcoming hardware support such as the Intel MPX extension [15]. To support this scenario, ASAP only needs to recognize instructions that may abort a program, and know their run-time cost.

ASAP is *easy to understand*. It uses a simple greedy algorithm to remove checks that are too costly. Developers can reason about the diminishing returns of additional safety checks, based on the intuition that 80% of the safety can be obtained with just 20% of the overhead (as §VI shows, the actual numbers are even better). We did consider more complex solutions before settling for this simplicity, though. For instance, ASAP could reason about dependencies between checks to obtain higher security. It could only remove checks that are provably not needed. Some possible extensions are discussed in §VII, but we believe the simplicity of the current solution to be a key strength.

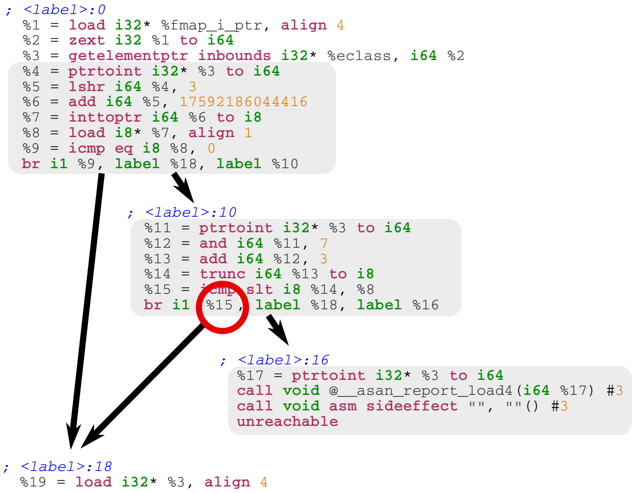
V. Implementation

This section presents the architecture of ASAP, and its core algorithms for detecting sanity checks, estimating their cost, and removing expensive ones from programs.

ASAP is based on the LLVM compiler framework and manipulates programs in the form of LLVM bitcode, a typed assembly-like language specifically designed for program transformations. It supports source-based instrumentation tools and those that have themselves been built on LLVM, which covers the majority of modern static instrumentation tools for C/C++/Objective C.

Users use ASAP through a wrapper script, which they invoke instead of the default compiler. In addition to producing a compiled object file, this wrapper also stores a copy of the LLVM bitcode for each compilation unit. This copy is used during subsequent stages to produce variants of the object with profiling code, or variants instrumented for a particular overhead budget.

ASAP works on programs one compilation unit at a time. It keeps no global state (except check data described later) and does not require optimizations at link-time. This is important for supporting large software systems that rely on separate and parallel compilation. The only phase in the workflow that requires a global view is the check selection phase, where ASAP computes a list of all sanity checks in the software system and their cost. This phase uses an efficient greedy selection algorithm described in §IV-A and has little impact on compilation time.


```
; <label>:0
  %1 = load i32* %fmap_i_ptr, align 4
  %2 = zext i32 %1 to i64
  %3 = getelementptr inbounds i32* %eclass, i64 %2
  %4 = ptrtoint i32* %3 to i64
  %5 = lshr i64 %4, 3
  %6 = add i64 %5, 17592186044416
  %7 = inttoptr i64 %6 to i8
  %8 = load i8* %7, align 1
  %9 = icmp eq i8 %8, 0
  br i1 %9, label %18, label %10

        ; <label>:10
          %11 = ptrtoint i32* %3 to i64
          %12 = and i64 %11, 7
          %13 = add i64 %12, 3
          %14 = trunc i64 %13 to i8
          %15 = icmp slt i8 %14, %8
          br i1 %15, label %18, label %16

                ; <label>:16
                  %17 = ptrtoint i32* %3 to i64
                  call void @__asan_report_load4(i64 %17) #3
                  call void asm sideeffect "", ""() #3
                  unreachable

; <label>:18
  %19 = load i32* %3, align 4
```

Fig. 4. A sanity check inserted by AddressSanitizer, in the LLVM intermediate language. The corresponding C code is `cc1 = eclass[fmap[i]]` and is found in `blocksort.c` in the bzip2 SPEC benchmark. Instructions belonging to the check are shaded. The red circle marks the branch condition which, when set to `true`, will cause the check to be eliminated.

ASAP automatically recognizes sanity checks. Recall from §IV-C that a sanity check verifies a safety property, aborts the program if the property does not hold, and is otherwise side-effect-free. ASAP searches for sanity checks by first looking at places where the program aborts. These are recognizable either by the special LLVM `unreachable` instruction, or using a list of known sanity check handler functions. The sanity checks themselves are the branches that jump to these aborting program locations. Figure 4 shows an example.

The listing is shown in the LLVM intermediate language, which uses static single assignment form (SSA); each line corresponds to one operation whose result is stored in a virtual register, numbered sequentially from `%1` to `%19`. This sanity check protects a load from the address stored in register `%3`. It computes the metadata address (`%7`), loads shadow memory (`%8`) and performs both a fast-path check (the access is allowed if the metadata is zero) and a slow-path check (the access is also allowed if the last accessed byte is smaller than the metadata). If both checks fail, the program is aborted using a call to `__asan_report_load4`.

ASAP computes the set I_c of instructions belonging to the check starting with the aborting function (`__asan_report_load4` in our example). It then recursively adds all operands of instructions in I_c to the set, unless they are also used elsewhere in the program. It also adds to I_c all branch instructions whose target basic block is in I_c. This is repeated until I_c reaches a fixpoint. In Figure 4, a shaded background indicates which instructions belong to I_c.

The instructions in I_c are used for computing check costs as described in §IV-A. A number of different profiling mechanisms can be used to measure instruction cost. Our choice fell on GCOV-style profiling counters, where the profiler uses one counter per basic block in the program and adds a counter increment before every branch instruction. Profiling thus determines the number of times each instruction was executed; we obtain an estimate of the actual cost by applying the static cost model for instructions that is built into LLVM's code generator. The advantage of this approach is that it is robust and yields cost estimates at instruction granularity that are unaffected by the profiling instrumentation itself.

ASAP removes checks that are too costly from the program by altering their branch condition. In our example in Figure 4, it replaces the branch condition `%15`, circled in red, by the constant `true`, so that the check can never fail. The rest of the work is done by LLVM's dead code elimination pass. It recognizes that all shaded instructions are now unused or unreachable and removes them.

All steps ASAP performs are generic and do not depend on any particular instrumentation. In fact, the ASAP prototype works for AddressSanitizer, SoftBound, UndefinedBehaviorSanitizer, and programmer-written assertions. It contains exactly four lines of tool-specific code, namely the expressions to recognize handler functions such as `__asan_report_*`. This makes it straightforward to add support for other software protection mechanisms. Also, we did not need to alter the instrumentation tools themselves in any way.

We mention one extra feature of ASAP that helped us significantly during development: ASAP can emit a list of checks it removes in a format recognized by popular IDEs. This makes it easy to highlight all source code locations where ASAP optimized a check. Developers can use this to gain confidence that no security-critical check is affected.

ASAP is freely available for download at http://dslab.epfl.ch/proj/asap.

VI. Evaluation

In our evaluation, we want to know both how fast and how secure instrumented programs optimized by ASAP are. Any software protection mechanism needs to quantify its overhead and security. More specifically in the case of ASAP, we ask:

1) Effectiveness: Can ASAP achieve high security for a given, low overhead budget? We show that ASAP, using existing instrumentation tools, can meet very low overhead requirements, while retaining most security offered by those tools.

2) Performance: How much can ASAP reduce the overhead of instrumentation on any given program? Does it correctly recognize and remove the expensive checks? What effect does the profiling workload have on performance? What are the sources of the residual overhead?

3) Security: Does ASAP in practice preserve the protection gained by instrumenting software? How many sanity checks can it safely remove without compromising security? We also analyze the distribution of both sanity checks and security vulnerabilities in software systems, and draw conclusions on the resulting security of instrumented programs.

A. Metrics

We quantify performance by measuring the runtime of both the instrumented program and an uninstrumented baseline and computing the *overhead*. Overhead is the additional runtime added by instrumentation, in percent of the baseline runtime. The *cost level* (see §IV-A) is determined from the minimum, maximum, and target overheads for a program, and the *sanity level* (see §III) is the fraction of static checks remaining in the program. To quantify the security of an instrumented program, we measure the *detection rate*, i.e., the fraction of all bugs and vulnerabilities that have been detected through instrumentation. The detection rate is relative to a known reference set of bugs and vulnerabilities (e.g., those detected by a fully instrumented program), because all bugs or vulnerabilities present in a particular software cannot be known in general.

B. Benchmarks and Methodology

We evaluated ASAP's performance and security on programs from the Phoronix and SPEC CPU2006 benchmarks, the OpenSSL cryptographic library, and the Python 2.7 and 3.4 interpreters. For instrumenting the target programs, we used AddressSanitizer (ASan) and UndefinedBehaviorSanitizer (UBSan) (described in §II), which are both widely applicable.

Unless otherwise noted, all performance numbers reported use cost levels that are *safe*, i.e., the optimized program is protected against all known vulnerabilities. Our default cost level is 0.01, for reasons described in the security evaluation in §VI-D.

We use a collection of real and synthetic bugs and vulnerabilities to quantify ASAP's effect on security. We also analyze to what degree ASAP affects the detection rate for the RIPE benchmark (a program containing 850 different combinations of buffer overflow exploits), known bugs in the Python interpreter, and the entries in the CVE vulnerability database for the year 2014.

The paragraphs below give details on setup, workloads and hardware used for each of our experiments.

1) SPEC CPU2006 Benchmarks: The SPEC CPU2006 suite is a set of 19 benchmark programs written in C/C++. Each program comes with a *training workload* that we used for profiling, and a *reference workload*, approximately 10× larger, used for measuring overhead.

We compiled for each program a baseline version without instrumentation and a fully-instrumented version. The runtime difference between these two is the overhead of instrumentation. In addition, we used ASAP to create optimized executables for cost level 0.01, and for sanity levels between 80% and 100%. We increased the resolution for sanity levels close to 100%, because small changes in the sanity level have a large impact on overhead in this region.

All the experiments were run for AddressSanitizer and UndefinedBehaviorSanitizer. For AddressSanitizer, we disabled stack trace collection to mimic a production scenario where performance is more important than informative debug output. We also turned off error recovery (UBSan will by default print a warning message and attempt to recover if some checks fail), choosing to always abort the program when an error is detected. Unfortunately, not all benchmarks are compatible with both instrumentation tools; we could run 14 benchmarks for ASan and 12 for UBSan.[1].

All data points have been measured on machines with a 3.4 GHz Intel Core i7 CPU (4 cores, hyperthreading) and 8 GB RAM.

2) OpenSSL: We compiled OpenSSL with AddressSanitizer. This is sufficient to protect the library from the Heartbleed vulnerability. To enable AddressSanitizer, only minor modifications to OpenSSL were needed: we changed `crypto/mem.c` to ensure that no custom memory allocators were used. We also compiled OpenSSL with `-DOPENSSL_NO_BUF_FREELISTS`, and disabled AddressSanitizer for the `OPENSSL_cpuid_setup` function because it contains incompatible inline assembly. We used the OpenSSL test suite as the profiling workload for the initial phase of ASAP.

To determine the instrumentation overhead, we measured OpenSSL's performance in a number of benchmarks. OpenSSL is most widely used in web servers; we benchmarked the throughput of a web server by measuring the number of pages that can be served per second. Our measurements use OpenSSL's built-in web server with a 3KB static HTML file. We also looked at the throughput of OpenSSL's cryptographic primitives, and the time it takes to run the test suite. OpenSSL performance measurements were done on a workstation with an Intel Xeon CPU (4 cores @ 2 GHz) and 20 GB RAM.

3) Python: We compiled the Python 3.4 interpreter with AddressSanitizer and UndefinedBehaviorSanitizer instrumentation. We obtained profiling data by by running Python's unit test suite. This same workload is used by the Ubuntu package maintainers for creating a profile-guided-optimization build of Python.

We evaluated performance using the default benchmarks from the Grand Unified Python Benchmark Suite [12]. All measurements were done on a workstation with an Intel Xeon CPU (4 cores @ 2 GHz) and 20 GB RAM.

C. Performance Results

We report the cost of security, with and without ASAP, in Figure 5. For each benchmark, we display three values: The overhead of full instrumentation (leftmost, dark bars), the overhead with ASAP at cost level 0.01 (gray bar, center), and the residual overhead (light bars, right). This data reveals a number of results:

Full instrumentation is expensive. On SPEC, both AddressSanitizer and UndefinedBehaviorSanitizer typically cause above 50% overhead.

ASAP *often reduces overhead to acceptable levels.* For eight out of 14 SPEC benchmark, ASAP reduces ASan overhead to below 5%. This result is also achieved for seven out of 12 benchmarks with UBSan. For three UBSan benchmarks, the overhead at cost level 0.01 is slightly larger than 5%.

For the remaining benchmarks, ASAP gives no security benefits because they are not elastic: their residual overhead is larger than 5%. In this case, ASAP can only satisfy the overhead budget by producing an uninstrumented program.

ASAP *eliminates most overhead due to checks.* In all cases except for `soplex`, the overhead at cost level 0.01 is very close to the residual overhead. Although many checks remain in the programs (87% on average for the benchmarks in Figure 5, generally more for larger programs such as Python), they do not influence performance much, because they are in cold code. These results show that ASAP correctly identifies and removes the hot checks.

ASAP*'s performance does not depend on precise profiling.* This is a corollary from the last observation. A bad profiling workload would not allow ASAP to identify the expensive checks, and thus lead to a large difference between overhead at $c = 0.01$ and residual overhead. Conversely, a perfect profiling workload can only improve ASAP's performance up to the residual overhead.

Even small reductions in security lead to large performance gains. In Figure 6, we show the speedups obtained when reducing the sanity level step by step. The gray area corresponds to the entire security-performance space that ASAP can navigate. The lightest gray area, or 47% of the total overhead, can be eliminated by removing just 1% of the sanity checks. This shows how additional cycles invested into security give diminishing returns, and confirms that indeed only few checks are hot.

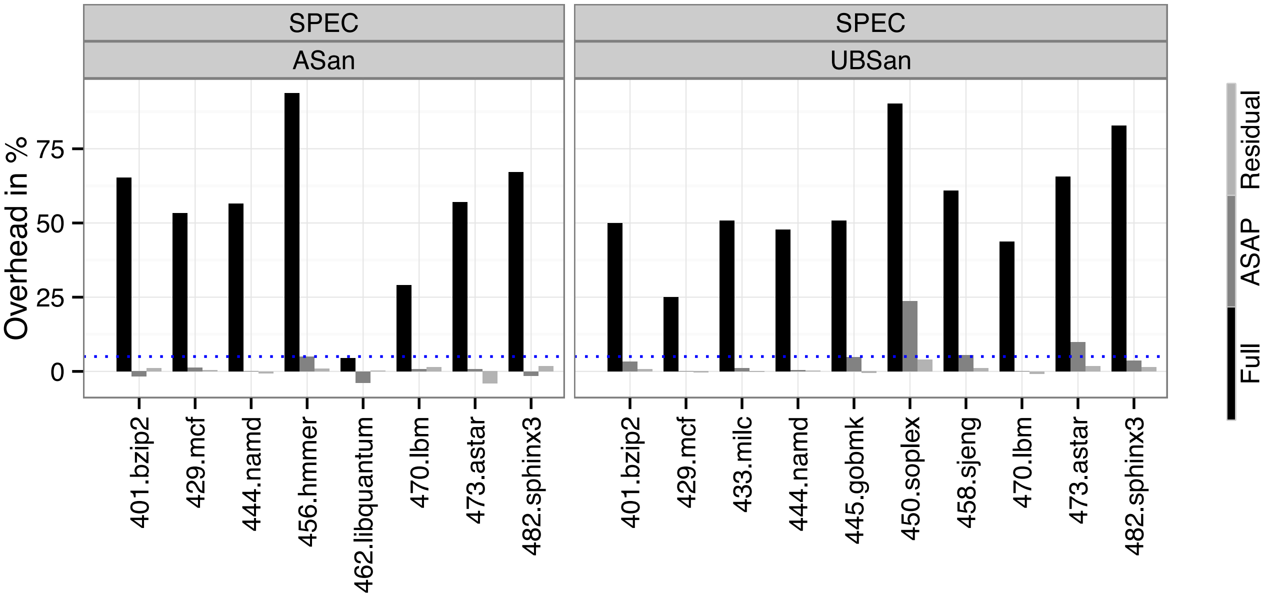
D. Security Evaluation

Developers and operators who use ASAP need to know how safe the resulting programs are. In particular, we measure how ASAP affects the detection rate of software instrumentation: what is the chance that a bug or vulnerability that was previously prevented by instrumentation is present in a ASAP-optimized program?

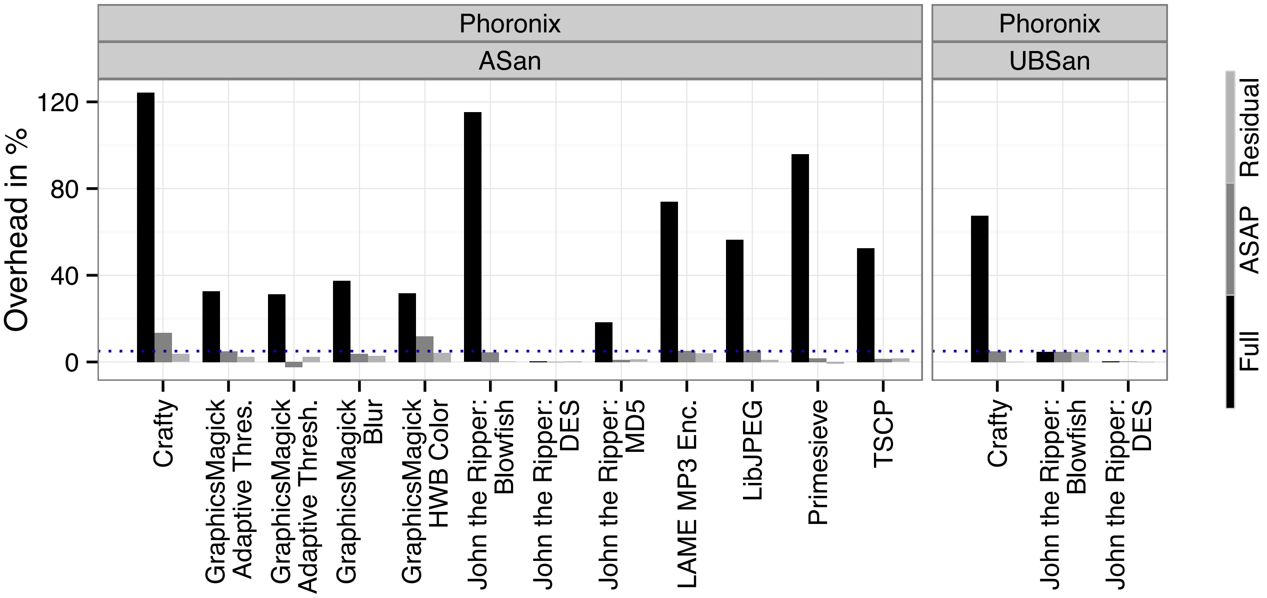
As discussed in §III on the sanity/performance trade-off, the detection rate depends primarily on the sanity level, i.e., the fraction of critical instructions that are protected with sanity checks. Since the sanity level is directly determined by the cost level, we can find an overall minimum cost level at which all known vulnerabilities would have been caught. The following paragraphs present our results of case studies on the OpenSSL Heartbleed vulnerability, Python bugs, and the RIPE

[1] Under ASan, `omnetpp` does not compile because it uses a custom `new` operator. `Xalancbmk` and `dealII` do not compile due to a bug in LLVM's cost model used by ASAP. `Perlbench` and `h264ref` abort at runtime due to buffer overflows involving global variables.

Under UBSan, 10 of 19 benchmarks abort because they perform undefined behaviors such as left-shifting a signed int by 31 places, multiplication overflow, or calling functions through function pointers of the wrong type. We could run 4 of them nevertheless, by selectively disabling one or two types of checks, and included them in the evaluation. We could not compile `omnetpp` with UBSan.



(a) ASAP performance results for SPEC benchmarks where $o_{min} < 5\%$.



(b) ASAP performance results for Phoronix benchmarks where $o_{min} < 5\%$.

Fig. 5. Summary of ASAP performance results. For each benchmark, we show three values: The darkest bar represents overhead for full instrumentation. The next bar shows overhead with ASAP at cost level 0.01. The lightest bar show the residual overhead, i.e., overhead that is due to other factors than sanity checks. Only elastic benchmarks (with residual overhead of less than five percent) are shown. ASAP brings the overhead of instrumentation close to the minimum overhead, while preserving a high level of security. For the benchmarks shown here, ASAP removes 95% of the overhead due to checks, and obtains an average sanity level of 87%.

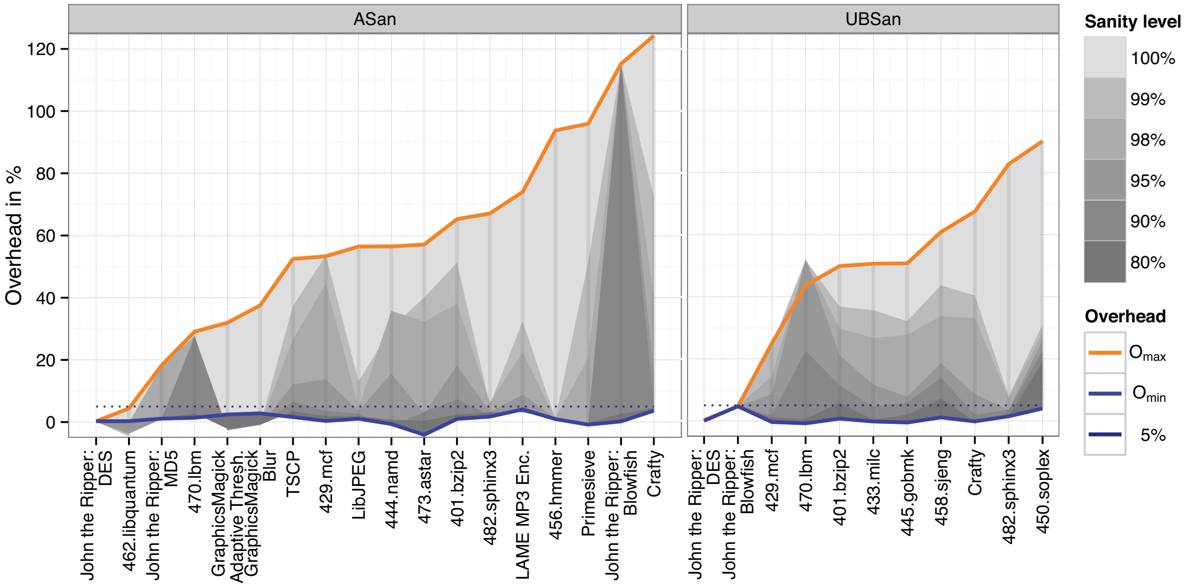


Fig. 6. This graph shows the space of possible performance-security trade-offs. The orange line shows overhead of existing instrumentation tools; it averages at 54% for ASan and 45% for UBSan. ASAP can reduce this overhead down to the blue minimal overhead line. The shade of the area corresponds to the sanity level (darker = fewer checks). Reducing the sanity level by a small value has a large impact on overhead; for example, reducing the sanity level to 99% reduces overhead by 47% on average. There are a few cases where programs with more sanity checks are slightly faster than programs with fewer checks (e.g., libquantum with ASan or lbm with UBSan). This is due to the sometimes unpredictable effects of checks on caches, compiler heuristics, optimizations etc.

benchmark; they demonstrate that a cost level of 0.01 would have been sufficient to prevent all vulnerabilities studied.

1) OpenSSL Heartbleed: The OpenSSL Heartbleed vulnerability is due to a bug in OpenSSL that manifests when processing heartbeat messages. Such messages have a length field and a payload, the size of which is expected to match the length field. Yet, attackers can send a heartbeat message with a length field larger than the payload size. When constructing the reply, the server would copy the request payload to the response, *plus whatever data followed it in memory*, up to the requested length. This allows the attacker to read the server's memory, including sensitive data like passwords.

The vulnerability can happen because the C programming language does not enforce memory safety. A pointer to the request packet can be used to read memory beyond the packet boundary, even though this memory belongs to different objects. The vulnerability is made worse because, for performance reasons, memory that is no longer used is not cleared. This means that the response returned to the attacker may contain not only data that is currently used, but also sensitive data from previous requests.

An attack that exploits the Heartbleed bug causes the OpenSSL program to read data past the bounds of the original request. Because of this, any instrumentation that detects overflowing memory reads will prevent the vulnerability. Indeed, compiling OpenSSL with AddressSanitizer produces a check that catches the overflow.

When we profiled OpenSSL using its test suite as profiling input, that critical check was never executed. This is because heartbeat messages are an optional and rarely used feature of OpenSSL, and the test suite does not cover them. This means that ASAP estimates the cost of the critical check to be zero and will never remove it, regardless of the target overhead specified.

We extended the test suite with a test case for heartbeat messages. Now the cost of the critical check is non-zero, but there are 15,000 other more expensive checks accounting for 99.99% of the total cost. We can further increase the check's cost by using larger payloads for the heartbeat messages we test. With a payload of 4KB, still 99.2% of the cost is spent in more expensive checks. Thus ASAP will preserve this sanity check for all cost levels larger than 0.008. This cost level corresponds to a target overhead that lies just slightly above the minimum overhead o_{min} of AddressSanitizer. It leads to only 5% reduction in throughput on a web server serving a 3kB web page via OpenSSL.

2) Python: The interpreter of the widely used Python scripting language consists of about 350 KLOC of C code, 1,900 of them assertions. When compiled with ASan instrumentation, the interpreter binary contains 76,000 checks.

We used the following methodology to evaluate the security of an ASAP-optimized Python interpreter: We started from the source code of the most recent 3.4 version of the language. Into this code, we inserted a set of bugs that have been present in earlier revisions; these form our reference set. Our criteria for choosing these bugs were (1) the bugs must be real-world problems recently reported on the Python issue tracker, (2) they must be detectable using instrumentation or assertions, and (3) they must be deterministically reproducible. We inserted the bugs by reverse-applying the relevant parts of the patch that fixed them.

The three bugs that we analyze are #10829, a buffer overflow in `printf`-style string formatting that ASan detects; #15229, an assertion failure due to an uninitialized object; and #20500, an assertion failure when an error occurs during shutdown.[2]

We ran the Python test suite as profiling workload and used the profiling data to generate an ASAP-optimized Python interpreter. Whenever the cost level is larger than 0.005, this interpreter is protected against all bugs that we analyzed. At cost level 0.01, the overhead of Python is at 55%, due to the large minimum overhead incurred from metadata handling in AddressSanitizer. §VI-E contains a more detailed evaluation of sanity checks and bugs in Python.

3) RIPE benchmarks: The RIPE benchmark suite [37] is a set of exploits for synthetic buffer overflows. It consists of a vulnerable program that attacks itself. In total, it features 850 unique attacks that differ in five characteristics: (1) the location of the buffer, e.g., on the stack or inside a structure; (2) the code pointer being overwritten, e.g., a return address; (3) whether the target pointer is overwritten directly or indirectly; (4) the type of shellcode; and (5) the function where the overflow happens, e.g., `memcpy` or `sprintf`.

The RIPE benchmark is well-known in the security community and contains a large number of exploits. However, its synthetic nature makes it problematic for evaluating ASAP: First, the exploits are all very similar; they differ only in few aspects of their construction, so that the number of effectively different scenarios is much smaller than 850. In particular, there are only ten distinct program locations where a memory corruption happens, so that the security gained by instrumentation is based on only ten sanity checks. Second, RIPE is designed for the sole purpose of overflowing buffers. There is no relevant workload that could be used for profiling. For the lack of an alternative, we exercised all the different overflow mechanisms to obtain profiling data. Third, RIPE makes strong assumptions about the compiler and the operating systems. Many exploits depend on the order of objects in memory, or on particular pointer values. Small changes in compilation settings or even between different runs of a program can cause such assumptions to fail; this makes it difficult to compare benchmarks.

For these reasons, we do not evaluate individual exploits in detail, and solely measure the minimal cost level needed to preserve the protection against buffer overflows gained by ASan instrumentation. ASAP preserves all critical sanity checks inserted by ASan for cost levels larger than 0.0004. Furthermore, nine out of ten buffer overflows happen inside library functions such as `memcpy`, which ASan redirects to its safe runtime library. Checks in ASan's runtime library are part of the residual overhead that ASAP does not yet address. ASAP currently preserves these checks at all cost levels.

4) Security Evaluation Summary: In our case studies on OpenSSL, CPython, and RIPE, we determined the minimum cost level to protect against all known vulnerabilities to be 0.008, 0.005, and 0.0004, respectively. We rounded this up to 0.01 and use this as default cost level for our performance experiments. A cost level of 0.01 corresponds to a sanity level of 94% in OpenSSL and 92% in CPython.

Note that a cost level of 0.01, even though it worked well in our experiments, does not imply that the resulting binaries are protected against all unknown vulnerabilities. Neither does such a cost level generalize to other software. Users of ASAP should analyze the result, e.g., by examining the elided checks as described in §V.

E. Discussion of Sanity Checks

To understand the security effect of ASAP, it is helpful to analyze the properties of sanity checks that are removed and preserved, respectively.

We first consider the 100 most expensive sanity checks in the Python interpreter. These checks together account for 29% of the total cost. They are in hot core locations of the interpreter: 49 of them belong to core Python data structures such as maps or tuples; 23 are in the main interpreter loop; 22 are in reference counting and garbage collection code; and six in other parts of the interpreter. Any meaningful Python program exercises the code where these checks reside. A bug in these parts of the interpreter would likely affect many Python scripts and thus be immediately detected. Hence we are confident that removing these checks in production is safe. The Python developers seem to partially agree with this: 6 out of these 100 checks are assertions in code regions that are only compiled when `Py_DEBUG` is defined, i.e., only during development.

In contrast, the checks that guard real-world bugs are executed rarely. The bugs in our case study are executed only (i) when a format string contains the `"%%"` character sequence, (ii) when a Python script circumvents the usual constructors and directly executes `__new__`, or (iii) when an error is raised during interpreter shutdown. We did not select the bugs to be particularly rare—it just happens to be that most real-world bugs *are* tricky corner cases.

Figure 7 sheds further light on this issue. For this graph, we looked at the checks in Python 2.7, and differentiate between checks that are located in buggy code, and "normal" checks. We take as buggy code those parts of the source code that have received bug fixes between the time Python 2.7 was released, until the current version 2.7.8.

We find that checks in buggy code are executed less frequently than regular checks. This makes them less likely to be affected by ASAP. For example, at cost level 0.01, ASAP removes 8% of all checks, but only 5% of the checks in buggy code. If we assume that our notion of buggy code

[2] Reports available on the Python bug tracker at http://bugs.python.org/

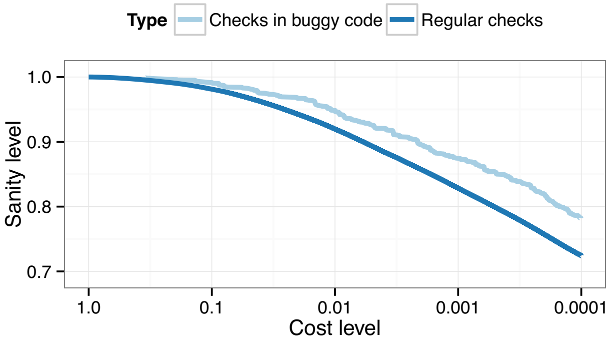


Fig. 7. Fraction of checks preserved by ASAP, for various cost levels. The dark line corresponds to the sanity level as computed by ASAP. The bright line corresponds to the fraction of protected buggy code. Because checks in buggy code have a lower cost on average than regular checks, they are more likely to be preserved.

is representative, we can conclude that the sanity level as computed by ASAP (92% in this case, for a cost level of 0.01) is a lower bound on the fraction of bugs that are protected by checks (95% in this case). This follows from the fact that the dark line is always below the bright line in Figure 7.

This experiment also shows that there *are* a few bugs in hot code, so using ASAP does reduce security. The computed sanity level gives developers an estimate of this reduction and allows them to make informed choices regarding the best trade-off between security and performance.

F. CVE Vulnerability Survey

We complete our security evaluation by studying known security vulnerabilities from the CVE database [23]. We focus on memory-related vulnerabilities because sanity checks are particularly promising for protecting against this category.

The CVE data set contains 879 memory-related vulnerabilities for the year 2014. For 180 of these, it was possible to obtain the source code and patch that fixed the vulnerability. From the source code and patch, we determined the location of the memory error itself. The error is not always located in the patched program part. For example, a common pattern is that developers add a missing check to reject invalid input. In this case, we searched for the location where the program accesses the illegal input and corrupts its memory. For 145 vulnerabilities, we could tell with sufficient certainty where the memory error happens.

We then manually analyzed the bugs to determine whether they lie in hot or cold parts of the program. We used four criteria to classify a code region as cold: (1) the code does not lie inside loops or recursively called functions, (2) the code is only run during initialization or shutdown, (3) comments indicate that the code is rarely used, and (4) the code is adjacent to much hotter regions which would dominate the overall runtime. In absence of these criteria, we classified a code region as hot.

Overall, we found 24 vulnerabilities that potentially lie in hot code regions. The other 121 (83%) lie in cold code where ASAP would not affect checks protecting against them. Because our criteria for cold code are strict, we think this is a conservative estimate. It provides further evidence that a large fraction of vulnerabilities could be prevented by applying instrumentation and sanity checks only to cold code areas,

The results of our CVE study are publicly available and can be accessed at http://dslab.epfl.ch/proj/asap.

VII. Extensions and Future Work

1) Elastic Instrumentation Tools: We believe there is a promising, yet unexplored area of building elastic instrumentation tools. This requires a change of mind: with techniques like ASAP, it is no longer the overall instrumentation overhead that matters, but the minimum, residual overhead when all checks are removed.

Builders of an elastic instrumentation tool take different design decisions. Consider, for example, the cost of a check in AddressSanitizer vs. SoftBound. SoftBound checks are more expensive because the metadata lookup is more complex. In contrast, a dynamic memory allocation is cheap for SoftBound because only a single lookup table entry needs to be updated, whereas AddressSanitizer needs to set up large shadow memory areas around the allocated memory object. Similar trade-offs exist for other operations such as function calls, memory de-allocation, or pointer arithmetic.

2) Other Sources of Overhead: With ASAP, we tackle the runtime overhead due to sanity checks. However, runtime overhead is not the only reason that prevents instrumentation from being used in practice. For example, systems where memory is the main bottleneck cannot afford to spend 15% of it for shadow memory. In other cases, performance might degrade due to registers being used for sanity checks, or cache lines being filled with metadata. The challenge in reducing this overhead is that the relationship between checks and metadata is complex. In most cases, it is not possible to predict statically where metadata will be used. Still we believe that it should be possible to gradually eliminate some of these other hurdles similarly to how ASAP deals with overhead from sanity checks.

3) Probabilistic and Dynamic Instrumentation: We are also considering a *probabilistic* version of ASAP. By default, ASAP uses a static cost threshold, above which a check is removed. It could alternatively remove checks probabilistically, with the probability proportional to a check's cost. An attacker who wanted to exploit a particular vulnerability then could not guarantee that it is exposed in the present instance of the program. Thus, the attacker risks that the attack is detected and that a zero-day vulnerability becomes known.

A probabilistic mechanism also enables collaboration between multiple users, or multiple machines in a cloud service. They could run software with different sets of sanity checks, in a way that further reduces overhead but causes vulnerabilities to be detected with high probability by at least one participant.

In a different scenario, users could use ASAP to build binaries at a range of cost levels. We could envision a system that dynamically switches between these binaries according to, for example, system load or the nature of requests. This leads to a system that automatically becomes more secure when resources are available.

VIII. Conclusion

We presented ASAP, a new approach to give developers control of how much runtime overhead they are willing to invest into adding security to their software systems. Our ASAP prototype automatically and selectively adds sanity checks to the software, making it as safe as possible for the chosen overhead.

The most expensive sanity checks lie in code that is frequently executed. However, exploits frequently target poorly tested and rarely executed code, where sanity checks are comparatively cheap. ASAP leverages this inverse relationship to prevent vulnerabilities from being exploited, while incurring only a low overhead that is suitable for production environments.

Acknowledgments

We thank Azqa Nadeem for her work on the experimental evaluation and CVE case study. We are thankful to Ed Bugnion, John Regehr, and our colleagues at the Dependable Systems Lab for ideas and helpful discussions. We would also like to thank the anonymous reviewers for their feedback. This work has been partially supported by ERC Starting Grant No. 278656 and by EPSRC grant EP/L022710/1.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS*, 2005.

[2] Qualys Security Advisory. Ghost: glibc gethostbyname vulnerability. http://www.openwall.com/lists/oss-security/2015/01/27/9.

[3] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *IEEE S&P*, 2008.

[4] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX ATC*, 2009.

[5] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *EuroSys*, 2013.

[6] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI*, 1994.

[7] Trishul M. Chilimbi and Matthias Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS*, 2004.

[8] Clang User's Manual. Undefined behavior sanitizer. http://clang.llvm.org/docs/UsersManual.html.

[9] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX ATC*, 1998.

[10] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: enforcing alias analysis for weakly typed languages. In *PLDI*, 2006.

[11] GCC coverage testing tool, 2010. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[12] Grand unified Python benchmark suite. https://hg.python.org/benchmarks/.

[13] Charles Antony Richard Hoare. Assertions: A personal perspective. In *Software pioneers*, pages 356–366. Springer, 2002.

[14] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided automated software diversity. In *CGO*, 2013.

[15] Intel Corporation. Intel architecture instruction set extensions programming reference. http://download-software.intel.com/sites/default/files/319433-015.pdf, 2013.

[16] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX ATC*, 2002.

[17] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG*, 1997.

[18] Baris Kasikci, Thomas Ball, George Candea, John Erickson, and Madanlal Musuvathi. Efficient tracing of cold code via bias-free sampling. In *USENIX ATC*, 2014.

[19] Samuel C Kendall. BCC: Runtime checking for C programs. In *USENIX ATC*, 1983.

[20] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity. In *OSDI*, 2014.

[21] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, 2004.

[22] Linux 2.6.7. NX (No eXecute) support for x86. https://lkml.org/lkml/2004/6/2/228, 2004.

[23] MITRE. Vulnerabilities and exposures. http://cve.mitre.org.

[24] Santosh Nagarakatte, Jianzhou Zhao, Milo M K Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *PLDI*, 2009.

[25] Santosh Nagarakatte, Jianzhou Zhao, Milo M K Martin, and Steve Zdancewic. CETS: Compiler enforced temporal safety for C. In *ISMM*, 2010.

[26] National Vulnerability Database. https://web.nvd.nist.gov/view/vuln/statistics, 2014.

[27] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *POPL*, 2002.

[28] Pwn2Own contest. http://www.pwn2own.com/.

[29] John Regehr. Use of assertions. http://blog.regehr.org/archives/1091, 2014.

[30] The Rust programming language. http://www.rust-lang.org/.

[31] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *NDSS*, 2004.

[32] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX ATC*, 2012.

[33] Skape. Preventing the exploitation of SEH overrides. http://www.uninformed.org/?v=5&a=2.

[34] Joseph L Steffen. Adding run-time checking to the portable C compiler. *Software: Practice and Experience*, 1992.

[35] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *IEEE S&P*, 2013.

[36] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *SOSP*, 2013.

[37] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: Runtime intrusion prevention evaluator. In *ACSAC*. ACM, 2011.

[38] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *OSDI*, 2014.